A Theory of Mediating Connectors to achieve Interoperability

Romina Spalazzese

To cite this version:

Romina Spalazzese. A Theory of Mediating Connectors to achieve Interoperability. Software Engineering [cs.SE]. Università degli studi de l'Aquila, 2011. English. NNT : . tel-00660816

HAL Id: tel-00660816
https://theses.hal.science/tel-00660816v1

Submitted on 17 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dipartimento di Informatica
Università di L'Aquila
SEA Group
Via Vetoio, I-67100 L'Aquila, Italy
http://www.di.univaq.it

PhD Thesis in Computer Science

A Theory of Mediating Connectors to achieve Interoperability

Romina Spalazzese

April 2011

Advisor
Prof. Paola Inverardi

© Romina Spalazzese, 2011. All rights reserved

To those who,
when facing troubles,
do not surrender:
they fall, but
pick themselves up
and fight.

ABSTRACT

Systems populating the Ubiquitous Computing environment are characterized by an often extreme level of heterogeneity at different layers which prevents their seamless interoperability. In this environment, heterogeneous protocols would cooperate to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other.
Although numerous efforts have been done in the literature, the automated and run-time interoperability is still an open challenge for such environment.

Therefore, this thesis focuses on *overcoming the interoperability problem between heterogeneous protocols in the Ubiquitous Computing*. In particular, we aim at providing a means to drop the interoperability barriers by automatically eliciting a way for the protocols to interact.
The solution we propose is the *automated synthesis of emerging mediating connectors* (also called mediators or connectors). Specifically, we concentrate our efforts to: (i) devising `AMAzING`, a process to synthesize mediators, (ii) characterizing protocol mismatches and related mediator patterns, and (iii) designing `MediatorS`, a theory of mediating connectors. The theory, and the process, are put in practice by applying them to a real world application, and have been adopted by the European Research Project CONNECT.

ACKNOWLEDGMENTS

The work is partly supported by the CONNECT European Project No 231167 of the Future and Emerging Technologies (FET) programme within the ICT theme of the Seventh Framework Programme for Research of the European Commission.

Obstacles are those frightful things you see when you take your eyes off your goal. (Henry Ford)

TABLE OF CONTENTS

Abstract i

Acknowledgments iii

Table of Contents vii

List of Figures ix

1 Introduction 1
- 1.1 Running Example: Photo Sharing Scenario 4
- 1.2 Formal Foundations 5
  - 1.2.1 Protocols as LTS 5
  - 1.2.2 Ontologies 8
- 1.3 Challenges and Contributions 9

2 Related Work 13
- 2.1 Emergent Computing 13
- 2.2 Mediator Patterns 15
- 2.3 Protocol Interoperability 16

3 The AMAzING Process 25
- 3.1 The Abstraction Phase 26
- 3.2 The Matching Phase 27
- 3.3 The Mapping Phase 28

4 Mediator Patterns 31
- 4.1 Photo Sharing Applications 32
- 4.2 A Pattern-based Approach for Interoperability Mismatches 34
- 4.3 Mediating Connector Architectural Pattern 34
- 4.4 Basic Mediator Patterns 37
- 4.5 Application of the Patterns to the Photo Sharing Scenario 44
- 4.6 Conclusion 46

5 MediatorS: a Theory of Mediating Connectors 47
- 5.1 Formalizing the Theory 47
  - 5.1.1 Abstraction Formalization 49
  - 5.1.2 Matching Formalization 53
  - 5.1.3 Mapping Formalization 55

5.2 Implementing the Theory: Algorithms . . . 57
5.2.1 Abstraction Algorithms . . . 58
5.2.2 Matching Algorithms . . . 61
5.2.3 Mapping Algorithms . . . 62
5.3 Correctness Discussion . . . 63
5.4 Conclusion . . . 66

6 Extending the MediatorS Theory to Encompass Middleware-Layer 67
6.1 Modeling Networked Systems . . . 68
6.1.1 Affordance . . . 69
6.1.2 Interface Signature . . . 69
6.1.3 Affordance Protocol . . . 71
6.2 Ontology for Mediation . . . 73
6.2.1 Middleware Ontology . . . 73
6.2.2 Application-specific Ontology . . . 77
6.3 Abstraction of Networked Systems . . . 78
6.4 Functional Matching of Networked Systems . . . 78
6.4.1 Affordance Matching . . . 79
6.4.2 Interface Mapping . . . 80
6.4.3 Behavioural Matching of Affordances . . . 82
6.5 Mapping of Networked Systems . . . 82
6.6 Conclusion . . . 82

7 Mediating Flickr and Picasa: a Case Study 85

8 The MediatorS Theory in CONNECT 91
8.1 Towards the CONNECT Architecture . . . 92
8.1.1 Heterogeneity Dimensions . . . 93
8.1.2 Beyond State of the Art Interoperability Solutions . . . 94
8.1.3 The CONNECT Architectural Framework . . . 96
8.1.4 Conclusion . . . 99

9 Discussion and Future Work 101

References 105

List of Figures

1.1 Abstract Networked Systems' model and the kind of interoperability . . . 3
1.2 Overview of the Photo Sharing scenario 4

3.1 Overview of the AMAzING Process 25
3.2 Abstraction phase of the AMAzING Process 26
3.3 Matching phase of the AMAzING Process 27
3.4 Mapping phase of the AMAzING Process 28

4.1 Peer-to-Peer-based implementation 33
4.2 Infrastructure-based implementation 33
4.3 Entities involved in a mediated system without and with Mediating Connector 36
4.4 Basic interoperability mismatches 38
4.5 Basic solutions for the basic mismatches 39
4.6 Variants of the Basic Mediator Pattern (1) 40
4.7 Variants of the Basic Mediator Pattern (2) 41
4.8 Variants of the Basic Mediator Pattern (3) 42
4.9 Variants of the Basic Mediator Pattern (4) 43
4.10 Behavioural description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 4.2 a) and P2P Photo Sharing version 1 of Figure 4.1 v1)) 45

5.1 An overview of our approach 48
5.2 Ontology mapping between Infrastructure-based Photo-Sharing Producer and the peer-to-peer Photo-Sharing version 1 (Figure 4.2 a and Figure4.1 v1 respectively) 49
5.3 The abstract protocol building 50
5.4 Abstracted LTSs of the Photo Sharing protocols 52

6.1 Infrastructure- and peer-to-peer-based photo sharing 72
6.2 Middleware ontology 74
6.3 Middleware alignment 75
6.4 Shared-memory based Photo Sharing after the mapping of middleware functions to reference middleware ontology 76
6.5 Middleware-agnostic peer-to-peer Photo Sharing 76
6.6 Photo Sharing ontology 77

7.1 Flickr and Picasa services 85
7.2 Flickr client protocol 86
7.3 Picasa server protocol 87
7.4 Ontological information 88
7.5 Abstracted protocols 89
7.6 Mediator between Flickr and Picasa protocols 90

8.1 Actors in the CONNECT Architecture 97
8.2 Networked System Model 97
8.3 The Discovery Enabler 98
8.4 A software Connector 99

CHAPTER 1

INTRODUCTION

Today's **Ubiquitous Computing** environment is populated by a wide variety of heterogeneous Networked Systems (NSs), dynamically appearing and disappearing, that belong to a multitude of application domains: home automation, consumer electronics, mobile and personal computing, to mention a few. Key technologies such as the Internet, the Web, and the wireless computing devices and networks can be qualified as ubiquitous, in the sense of Mark Weiser [121], even if these technologies have still not reached the maturity envisioned by the Ubiquitous Computing and the subsequent pervasive computing and ambient intelligence paradigms because of the extreme level of heterogeneity of the underlying infrastructure which prevents seamless interoperability.
The Ubiquitous Computing (UbiComp) was proposed by Mark Weiser in the Nineties [122] [121] as the direction for technology development in the twenty-first century. But the early basics for this new philosophy were created in 1988 as "tabs, pads and boards" [123]. A further evolutionary step that followed the UbiComp is represented by those made for distributed systems and mobile computing [103]. The Ubiquitous vision was based on the principle of making the computer able to vanish in the background. Weiser defined UbiComp as the method to increase the use of computers making available many of those present in the physical environment in an efficient and invisible manner to users. The term Ubiquitous Computing, hence refers to that trend that sees the interaction between user and a dynamic set of devices and networked services,which are often invisible and embedded into objects in the environment. This trend fosters an environment in which the distinction between physical and digital devices is not clear. All entities in a physical space are integrated in a cohesive programmable unit to create, for example, smart offices and active spaces.
The environment includes not only a large set of devices, spanning powerful computers and workstations, but also interaction and communication patterns for users to support user-oriented services since the UbiComp suggests the ability for users to enter the environment in a natural way, without being aware of who or what populate it. Furthermore, the user should be able to use the services available using its devices without complex procedures and manual configurations. This leads to the need for high quality reliable services possibly partially present on devices. As we already said, the currently available technologies and computations have not yet reached the maturity required by the ubiquitous paradigm. The ubiquity of digital systems is not complete due to the fact that it is still tied and dependent on technology: the effectiveness of the composition and integration of

networked systems, i.e., is proportional to the level of interoperability of the underlying technologies.
All the variety of resources typical of the ubiquitous vision leads to the creation of technological islands of interoperating systems and requires the construction of interoperability bridges among different islands to make their communication possible. In addition, the speed with which technology changes at each level of abstraction, increasingly shortens the life of these interoperability bridges making increasingly complex their maintenance process. The availability of a large number of heterogeneous resources, leads to a paradox: to be able to exploit a potentially infinite number of capabilities and services implies the need to abandon any predetermined knowledge. In other words, applications should be able to interpret and learn what the environment offers, and dynamically adapt to it to achieve their aims while making minimal assumptions.
The described Ubiquitous vision puts emphasis on the activities that one wants to play rather than the means used to do this and sees computers evanish into the background. This vision perfectly fits our idea of building an interoperability solution. That is, each entity should maintain its own characteristics, being able to communicate and cooperate with the others despite diversities. In particular, heterogeneous protocols would cooperate to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other.

The term **protocol** refers to *interaction protocols* or *observable protocols*. That is, a protocol is the behaviour of a system in terms of the sequences of messages (input/output actions) visible at the interface level, which it exchanges with other systems. In other words an interaction protocol provides a system behavioural description taking an external (black-box) perspective. More specifically, in this thesis the term protocol will first refer to *application-layer* protocols. In a second step this notion will be extended to include also *middleware-layer* protocols, i.e., subsequently we will consider together both application- and middleware layer protocols.

We further focus on **compatible protocols** or **functionally matching protocols**. Compatibility or functional matching means that heterogeneous protocols can *potentially communicate* by performing *complementary sequences of actions* (or *complementary conversations*).
Potentially means that communication may not be achieved because of **mismatches** (heterogeneity), i.e., the languages of the two protocols are different, although semantically equivalent. For example, protocol languages can have: (i) different granularity, or (ii) different alphabets. Protocols behavior may have, for example, different sequences of actions because of (a.1) the order in which actions are performed by a protocol is different from the order in which the other protocol performs the same actions; (a.2) interleaved actions related to *third parties communications* i.e., communications with other systems, the environment. In some cases, as for example (i), (ii) and (a.1), it is necessary to properly perform a manipulation of the two languages to enable communication. In the case (a.2) it is necessary to provide an abstraction of the two actions sequences that results in sequences containing only actions that are relevant to the communication.
Communication is then possible if the two possibly manipulated (e.g., reordered) and ab-

stracted sequences of actions are complementary, i.e., are the same sequences of actions while having opposite output/input "type" for all actions.
For the sake of simplicity, and without loss of generality, we limit the number of protocols to two but the work can be generalized to an arbitrary number of protocols.

Although numerous efforts has been done in the literature, automated and run-time interoperability is still an open challenge for such environment where protocols do not know their possible interlocutors a priori.
Therefore, the *problem* we address and overcome, is the *interoperability between heterogeneous protocols in the Ubiquitous Computing environment*.

With **interoperability**, we mean the ability of heterogeneous protocols *to coordinate to reach their goal(s), i.e., communicate*. The coordination is expressed as *synchronization*, i.e., two systems succeed in communicating if they are able to synchronize reaching their goal(s).
In this thesis we will formulate and provide a solution for the *application-layer interoperability*. However the approach we propose is general, and then subsequently we will extend it by considering together *application-layer* and *middleware-layer interoperability*. Figure 1.1 illustrates the abstract Networked Systems' model and the kind of interoperability we consider in this thesis.

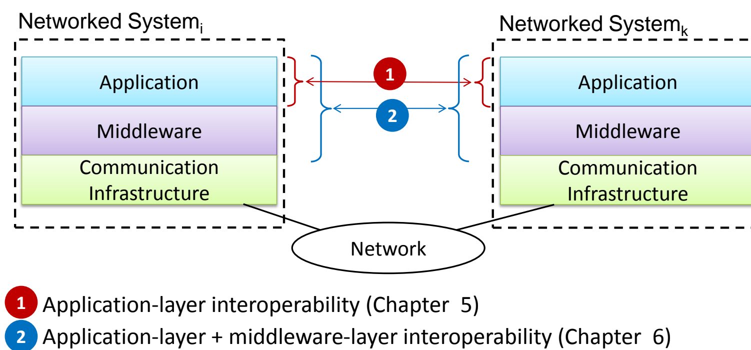


Figure 1.1: Abstract Networked Systems' model and the kind of interoperability

Summarizing, we want to automatically achieve interoperability between functionally matching protocols. Based on the terminology explained above, the **research question** that this thesis addresses can be formulated as follows:

"given compatible protocols, is it possible to automatically synthesize a mediator which enables them to communicate (solving their mismatches)?"

To answer this question, i.e., in order to make communication between heterogeneous protocols possible, here we propose as **solution** *a theory of mediators*. The theory, reasoning about the mismatches of the compatible protocols, automatically elicits and synthesizes an *emerging mediator* that solves them allowing protocol interoperability. The theory paves the way for run-time (or on-the-fly) approaches to the mediators synthesis.

A **mediator** is then a protocol that allows the communication among compatible protocols by mediating their differences.

In the reminder of this chapter we: introduce a running example (Section 1.1) that will be also used in the subsequent chapters for explanation purpose; characterize protocols as Labelled Transition systems 1.2.1; and illustrate the challenges and contributions of this thesis (Section 1.3), we provide its outline and we list the papers on which it is based.

1.1 Running Example: Photo Sharing Scenario

In order to illustrate our approach to mediator synthesis, we consider the simple yet challenging scenario of Photo Sharing within a public space such as a stadium which is illustrated in Figure 1.2.

Figure 1.2: Overview of the Photo Sharing scenario

The target environment allows both Infrastructure-based (IB) and ad hoc peer-to-peer (P2P) Photo Sharing. In the IB 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 P2P 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 his/her 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 behaviour. In particular, the RPC interaction paradigm suits quite well the IB service, while a distributed shared data space is more appropriate for the P2P 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.

The case we are addressing deals with protocols that are willing to communicate to reach a common goal, e.g., photo sharing, but cannot communicate because of protocol mismatches. For instance, as mentioned above, protocols may have different alphabets, i.e., different action labels, or the order in which actions are performed by protocols may be different. Our approach automatically creates a mediator (protocol) that solves the mismatches which prevent protocol communication by suitably mediating such differences, thus enabling the communication among protocols.

Note that due to the limited capacities of the devices, the creation process of the mediator can be performed, for instance, by a supporting infrastructure made up by more powerful computers. One can think that coming devices advertise their applications behaviors and ontological characterizations to the supporting infrastructure which starts computing the mediators -if possible. Then, when the will to communicate is manifested, the mediator -if it exists- can be either ready or partially computed. While the mediator could not exist because applications are not compatible and then do not have a way to communicate. The supporting infrastructure could then be helpful for both the automated and on-the-fly synthesis of mediators.

1.2 Formal Foundations

The application-layer interaction protocol, as described in the previous sections, is the behaviour of a system in terms of the actions it exchanges with other application-layer protocols. We recall that an interaction protocol describes the behaviour of a protocol from an external point of view thus consisting of the sequence of input and output actions at interface level. Further, in order to realize protocol mediators, ontologies play an important role.
In Section 1.2.1 we provide a characterization of protocols (i.e., behaviours) using Labelled Transition Systems (LTSs) [70] while in Section 1.2.2 we describe ontologies. Note that the in the following we neither provide an algebra nor a language while we express in a syntactic notation how the mediator is built.

1.2.1 Protocols as LTS

LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioural languages such as process algebras. Let *Act* be the set of observable actions (input/output actions), we get the following definition for LTS:

Definition 1 (*LTS*) *A LTS P is a quadruple* (S, L, D, s_0) *where:*
S *is a finite set of states;*

$L \subseteq Act \bigcup \{\tau\}$ is a finite set of labels (that denote observable actions) called the alphabet of P. τ is the silent action. Labels with an overbar in L denote output actions while the ones without overbar denote input actions. We also use the convention that for all $l \in L, l = \bar{\bar{l}}$[1].
$D \subseteq S \times L \times S$ is a transition relation;
$s_0 \in S$ is the initial state.

We then denote with $\{L \bigcup \{\tau\}\}^*$ the set containing all words on the alphabet L. We also make use of the usual following notation to denote transitions:
$s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$

We consider an extended version of LTS, where the set F of the LTS' *final states* is explicit. An **extended LTS** is then a quintuple (S, L, D, F, s_0) where the quadruple (S, L, D, s_0) is a LTS and $F \subseteq S$. From now on, we use the terms LTS and extended LTS interchangeably, to denote the latter one.

The initial state together with the final states, define the boundaries of the protocol's coordination policies. A **coordination policy** is indeed defined as any trace that starts from the initial state and ends into a final state. It captures the most elementary behaviours of the networked system which are meaningful from the user perspective (e.g., upload of photo of photo sharing producer meaning upload of photo followed by the reception of one or more comments). Then, a coordination policy represents a communication (i.e., coordination or synchronization) unit. We get the following formal definition of traces/coordination policy:

Definition 2 (*Trace* or *Coordination Policy*) *Let $P = (S, L, D, F, s_0)$.*
A trace t of P is a sequence $t = l_1\ l_2\ \ldots\ l_n \in L^*$ such that:
$\exists (s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2\ \ldots\ s_m \xrightarrow{l_n} s_n)$ *where* $\{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F$.

We use the usual compact notation $s_0 \stackrel{t}{\Rightarrow} s_n$ to denote a trace, where t is the concatenation of actions of the trace.
Moreover we define a **subtrace** as any (sub)sequence of a trace of a protocol (it may be also a trace). More formally:

Definition 3 (*Subtrace*) *Let $P = (S, L, D, F, s_0)$ and $t = l_1\ l_2\ \ldots\ l_n \in L^*$ be a trace of P.*
A subtrace st of t is a sequence $st = l_i l_{i+1} \ldots l_j \in L^*$ such that:
$\exists (s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2}\ \ldots\ s_j \xrightarrow{l_j} s_k)$ *where* $\{s_i, s_{i+1}, s_{i+2}, \ldots, s_j, s_k\} \in S$

Similarly to traces, also in this case we use the compact notation $s_i \stackrel{st}{\Rightarrow} s_k$.
LTSs can be combined using the LTS parallel composition operator. Several semantics

[1] This convention comes from *Calculus of Communicating Systems* (CCS) [79]

have been given in the literature for this operator. In the following we provide a description of parallel composition we need that is similar to the one of CSP (*Communicating Sequential Processes*) [101]. More precisely, in our case protocols P and Q synchronize on complementary actions while proceeding independently when engaged in non complementary actions. Actions are complementary if they are the same action while having opposite output/input type. Note that we do not need choice operators because the way in which the parallel is defined is non-deterministic. Moreover, we need a *synchronous* reference model as the one of CSP or FSP (*Finite State Process*) [76] where the synchronization is forced when an interaction is possible by requiring simultaneous participation of P and Q. Differently, the asynchronous model like the one of CCS (*Calculus of Communicating Systems*) [79], would allow agents to nondeterministically choose to not interact by performing complementary actions a and $\overline{a}$ separately.

Although the semantics and the model we need are *à la* CSP, we use CCS because (i) it is able to emulate the synchronous model of CSP thanks to the restriction operator and (ii) it has several characteristics that CSP does not have and that we need, e.g., complementary actions and τ .

Then our parallel composition semantics is that protocols P and Q synchronize on complementary actions producing an internal action τ in the parallel composition. Instead, P and Q can proceed independently when engaged in non complementary actions. An action of P (Q resp.) for which no complementary action exists in Q (P resp.), is executed only by P (Q resp.), hence, producing the same action in the parallel composition.

Definition 4 (*Parallel composition of protocols*) *Let* $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q,\ L_Q,\ D_Q,\ F_Q,\ s_{0_Q})$. *The parallel composition of* P *and* Q *is defined as the LTS* $P|Q = (S_P \times S_Q,\ L_P \cup L_Q,\ D,\ F_P \cup F_Q,\ (s_{0_P}, s_{0_Q}))$ *where the transition relation* D *is defined as follows:*

$$\frac{P \xrightarrow{m} P'}{P|Q \xrightarrow{m} P'|Q} \qquad (where\ m \in L_P \wedge \overline{m} \notin L_Q)$$

$$\frac{Q \xrightarrow{m} Q'}{P|Q \xrightarrow{m} P|Q'} \qquad (where\ m \in L_Q \wedge \overline{m} \notin L_P)$$

$$\frac{P \xrightarrow{m} P';\, Q \xrightarrow{\overline{m}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (where\ m \in L_P \wedge \overline{m} \in L_Q)$$

Note that we build the parallel composition of protocols P and Q, with their environment, i.e., other systems protocols E, and a mediator M, i.e., a protocol that mediating P and Q differences allows their communication. The resulting composed protocol $P|Q|E|M$ is restricted to the language made by the union of the common languages between each

pair of protocols. Thus, this restriction forces all the protocols to synchronize when an interaction is possible among them.

1.2.2 ONTOLOGIES

Ontologies play an important role in realizing connectors which primarily relies on reasoning about protocol functionalities. More in detail, what is needed is to identify matching sequences of observable actions among the actions performed by the protocols. Ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of ubiquitous networked systems. Indeed, “an ontology is a formal, explicit specification of a shared conceptualization” [114]. 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 [46]. 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 [10].

In the literature, [67, 66] ontologies ,and ontology mapping, are presented as logical theories as follows:

> “an *ontology* is a pair $O = (S, A)$, where S is the (ontological) signature describing the vocabulary and A is a set of (ontological) axioms specifying the intended interpretation of the vocabulary in some domain of discourse”.
>
> “A *total ontology mapping* from $O_1 = (S_1, A_1)$ to $O_2 = (S_2, A_2)$ is a morphism $f : S_1 \rightarrow S_2$ of ontological signatures, such that, $A_2 = f(A_1)$, i.e., all interpretations that satisfy O_2’s axioms also satisfy O1’s translated axioms”.

We specialize the above ontology mapping, that maps single elements of S_1, by defining an *abstraction ontology mapping* where the f is such that maps the S_1 language (i.e., S_1^*) into S_2, i.e., $f : S_1^* \rightarrow S_2$.
We use such specialized ontology mapping on the ontologies of the compatible protocols, where the vocabulary of the source ontology is represented by the language of the protocol. More formally:

Definition 5 (*Abstraction Ontology Mapping*) *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *be a protocol,*
- $O_P = (L_P^*, A_P)$ *be the ontology of* P*,*

- *$O = (L, A)$ be an ontology,*
- *$st \in L_P^*$ be a subtrace of a trace of P.*

The abstraction ontology mapping is a function $maps$ such that:
$maps : L_P^* \rightarrow L$.

The application of the above abstraction ontology mapping $maps$ on the ontology of P returns as result the abstract ontology $L_{abs} = \{l \in L : \forall\ st \in L_P^*\ \ l = maps(st)\}$.

Towards enabling connectors, in Chapter 5 we will use application ontologies characterizing the application actions. Subsequently, to support the theory extension, we will also extend such ontological description by introducing a middleware ontology that is the basis for middleware protocol mediation (Section 6.2.1) together with domain-specific application ontologies characterizing application actions defining both control- and data-centric concepts (Section 6.2.2).

1.3 Challenges and Contributions

As already pointed out, this thesis focuses on the interoperability problem between heterogeneous protocols and the solution we propose is a theory to automatically synthesize mediators allowing protocols to interoperate by solving their mismatches.
The interoperability problem and the notion of mediator to solve it are not new. They have been the focus of extensive studies within different research communities (this is widely described in Chapter 2). Protocol interoperability come from the early days of networking and different efforts, both theoretical and practical, have been done to address it in several areas including, for example: protocol conversion, component adaptors, Web services mediation, theories of connectors, wrappers, bridges, and interoperability platforms.
Although the existence of numerous solutions in the literature, to the best of our knowledge, all of them either: (i) assume the communication problem solved (or almost solved) by considering protocols already (or almost) able to interoperate; or (ii) are informal making it impossible to be automatically reasoned; or (iii) deal with the interoperability separately either at application layer or at middleware layer; or (iv) follows a semi-automatic process for the mediator synthesis requiring a human intervention; or (v) consider only few possible mismatches.

The list above results in limitations making it difficult, if not impossible, to automatically synthesize mediators in the context we consider, i.e., where protocols have no prior knowledge one another and the way to achieve communication -if it is possible- needs reasoning to emerge. Therefore, informally we can say that our work puts the emphasis on “eliciting a way to achieve communication -if communication is possible” while it can gain from more practical treatments of the mediators (or also of converters, or adaptors

or coordinators) synthesis. Our work has been devoted in particular to the *elicitation and definition of a comprehensive mediator synthesis process and of a theory of emerging mediators* which also includes *related supporting methods and tools*. In particular, our work has led us *to design automated techniques and tools to support the devised synthesis process*, from protocol abstraction to matching and mapping.
In the following, we summarize the challenges that we investigated and the related results achieved, that are further detailed in the next chapters, which are *contributions* of this thesis.

- **Devising a comprehensive mediator synthesis process.**
  We defined AMAzING, a process which: (i) performs an abstraction of the protocols (ii) checks their compatibility and (iii) in case of successful check (i.e., the protocols are compatible), automatically synthesizes a mediator while returns an incompatibility message otherwise.

- **Approaching the mediator synthesis problem in a systematic way by adopting a pattern-based solution.**
  We have precisely characterized the *protocol mismatches* that we intend to solve with our connectors, as well as the basic *mediator patterns* that solve the classified problems. We believe that this classification of mediator patterns can serve, in the future, as a basis for addressing compositional mediator synthesis.

- **Designing and extending a theory of mediators.**
  We elaborated MediatorS, a theory of mediating connectors (also referred as mediators or connectors), which defined the matching and mapping relations over the interaction behaviours of applications abstracted as Labelled Transition Systems. The theory has been applied to several case studies including Instant Messaging real world applications.
  Further, we have revised and extended the theory, so to deal also with middleware layer protocols and conveyed data.

THESIS OUTLINE

As detailed in the following chapters, significant progress has been achieved towards actually enabling the connection of networked systems despite heterogeneity. The reminder of the thesis is organized as follows.
Chapter 2 surveys the research work related to the topics of this thesis and, by making a comparison with them, positions our work with respect to the literature.
The first contribution of the work, i.e. the AMAzING process is presented in Chapter 3 describing three phases: protocol abstraction, matching and mapping.
Chapter 4 illustrates another contribution, the mediator patterns, mades up by a set of design building blocks to tackle in a systematic way the protocol interoperability problem.
The core contribution is described in Chapter 5, where a formalization of the MediatorS

theory is given, under some assumptions, together with its implementation in terms of algorithms.
Subsequently, in Chapter 6, we describe another contribution by extending the theory presented in the previous chapter in order to deal with a more complete protocol description. That is, in addition to application-layer protocols considered in Chapter 5, it takes into account also middleware-layer protocols and data.
We validate our theory, and hence process, on a real world case study in Chapter 7 and we report in Chapter 8 about the CONNECT European Project that adopted both the process and the theory.
Finally, we conclude in Chapter 9 with a discussion about the obtained achievements and by outlining future work perspectives.

PUBLICATIONS

Some of the contributions listed above are also reported in the following published papers on which this thesis is based:

- the paper [109] illustrates a preliminary version of the mediator patterns;
- the work [108] extends and revises the patterns presented in the previous paper;
- the paper [107] gives an overview of the problem and approach of this thesis;
- the paper [110] provides an high-level view of the theory for application-layer;
- the work [62] describes more in detail the theory providing a formalization of it;
- the paper [18] shows a combined approach between the mediator synthesis and the monitoring mechanism;
- the work [15] presents first results towards the software architecture of the CONNECT project.

Furthermore, we are planning a number of works according to the future directions described in the final discussion of this thesis.

CHAPTER 2

RELATED WORK

This Chapter gives an overview about the works in the literature related to this thesis. Moreover, a comparison with the existing works is provided all along the text thus positioning our work with respect to the literature. The works description is organized based on the different research areas and aspects they belong to, that are: Emergent Computing (Section 2.1), Mediator/Connector Patterns (Section 2.2), and Protocol Interoperability (Section 2.3).

2.1 EMERGENT COMPUTING

The Emergent Computing (EC) is described as a highly complex process which arises from the cooperation of many simple processes. This paradigm is inspired by and explores biological and social systems in which the behaviour of complex global level emerges in a nonlinear fashion from a large number of interactions between low-level components [102]. The Emergent Computation is a type of bottom-up computation in the sense that the system behaviour is neither globally nor fully programmed but simply emerges as an aggregation of local information [68].

Emergent Computing Systems are defined by Olson et al. as systems composed of independent agents that behave according to explicit instructions [88, 87]. The system shows spatial and/or time-implicit patterns arising as result of interactions between the subcomponents that constitute it and/or between them and their environment. Patterns belong to a higher level with respect to that of agents and are not explicitly coded in their specifications.

Among the works belonging to the Emergent Computing research we can mention [115]: (i) that of John von Neumann's on self-reproducing automata (1940) in systems with computational abilities of a universal Turing machine; (ii) a work based on Genetic Algorithms which are computational models of evolution and growth and where the central concept is the Evolution by Natural Selection; (iii) work about classifier systems, which can be seen as an evolution of genetic algorithms, and which can be thought as super-agents that combine different genetic algorithms; and finally (iv) neural networks that are a complex array of layers designed to mimic the paths of neurons in the brain.

In the EC context, one line of work that has correlation with the work that we propose in

this thesis, is the one of Aberer et al. [90, 4, 3]. Their study, dating back to the beginning of 2000, analyzes the principles and conditions under which global properties emerge from local interactions. The context of their study is the Semantic Web, that is highly dynamic, in which agents (peers) do not necessarily have the same vision. The idea is to apply such principles to an interpretation of structured data.

These peer dynamically build information and knowledge creating new semantic community. They establish a new form of semantic interoperability namely Emergent Semantics (ES) which is based on dynamic agreements. This latter is an emerging phenomenon built incrementally. Its state at a given instant of time depends on the frequency, the quality, and the efficiency with which negotiations can be conducted to achieve an agreement on a common interpretation in the context of a specific purpose. In the following we report the key principles of the Emergent Semantics [4].

1. *Agreement as a Semantic Handshake Protocol* The set of mutual beliefs among interacting agents is called the agreement or consensus. Meaningful exchanges can happen only on the basis of mutually accepted propositions. The strength of the agreement will depend on the strength of the accepted propositions, their quality and trustworthiness.

2. *Agreements emerge from negotiations* The information exchange between agents is necessary to negotiate new agreements and to verify the existent ones. When an agent learns more about other agents, and when the interests are spread or become more focused, then the agreements evolve.

3. *Agreement emerge from local interactions* An assumption on the Emergent Semantics is that the commitments are dynamic and incrementally established. A challenge is the scalability because of the complexity of the ES and the costs of communication. The practice suggests as a solution to use Emergent Semantics locally to reduce communication costs and to derive global agreements through aggregation of local agreements.

4. *Agreements are dynamic and self-referential approximations* Since the interpretation can depend on the context, the agreements are established dynamically and the local consensus is influenced by the existing global agreements, thus the process of establishing agreements is self-referential.

5. *Agreements induce semantic self-organization* Considering the dynamics and self-referential nature of emergent semantics, it is not far-fetched to view it as the result of a self-organization process.

According to what described, there is a correlation between the model of Emergent Semantics and our work. Similarly to ES, in our theory we have the concept of *agreement* that is represented by the ontology mapping and that plays a crucial role. Indeed, we build upon it for the mediator construction.

Moreover, such agreement is *local* meaning that it relates to potentially communicating protocols. Thanks to it we are able to automatically derive a mediator that is the fundamental (missing) piece for the protocols to constitute a global system where protocols interoperate.

2.2 Mediator Patterns

In the last decades, protocol mediation has been investigated in several contexts, among which integration of heterogeneous data sources [126, 125] and design patterns [54]. Indeed, an approach to protocol mediation is to categorize the types of protocol mismatches that may occur and that must be solved in order to provide corresponding solutions to these recurring problems. This immediately reminds of patterns whose pioneer was Alexander [5]: "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Patterns have received attention in several research areas.

In the **software architecture** field [92, 106, 116], Bushmann *et al.* [28] gave the following definition: "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. [...] An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them". More recently, architectural patterns have been revisited in [9], which proposes a pattern language.

The "gang of four" in [54] have defined **design patterns** as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". Among all, two design patterns are related to ours: the *Mediator Design Pattern* that is behavioral, and the *Adapter Pattern* that is structural. The former is similar because it serves as an intermediary for coordinating the interactions among groups of objects but it is different because its main aim is to decrease the complexity of interactions. The latter is similar because it adapts the interfaces of the objects while it differs because our mediator is not just an interface translator.

In the **Web services** context, several works have introduced basic pattern mismatches and the corresponding template solutions to help the developers to compose mediators [14, 34, 113, 129, 65]. In particular, references [129, 14] are related to our work since they identify and classify basic types of mismatches that can possibly occur when compatible but mismatching processes try to interoperate. Moreover, they provide support to the developers by assisting them while identifying protocol mismatches and composing mediators. In [129], the authors also take into consideration more complex mediators

obtained by composition of basic ones. The main difference between these two works and ours is the semi-automation issue. Indeed, they require the developer intervention for detecting the mismatches, configuring the mediators, and composing basic mediators while, thanks to formal methods as we will illustrate in the following chapters, we aim at automatically deriving the mediator under some conditions.
Reference [45] presents an algebra over services behavioral interfaces to solve six mismatches and a visual notation for interface mapping. The proposed algebra describes with a different formalism, solutions that are similar to our basic patterns and this can be of inspiration for us in the direction of reasoning.

The work [130] proposes an **adaptor** theory for the interoperability of applications. Applications are assumed to agree on the ordering of messages and the theory does not concern the messages buffering, thus not solving ordering, splitting and merging mismatches.

In the **connector** area [105], Spitznagel in her Ph.D. thesis [111] illustrates a set of patterns of basic connector's transformations that is adaptations, enhancements. She also shows how these patterns can be applied compositionally to simple connectors in order to produce a number of more complex connectors. In particular she shows how to select and to apply such transformations in the domain of dependability and proposes a prototypal tool to semi-automatically derive new connectors as enhancements.
The work [124], exploiting concepts from the Software Architectures and from the Category Theory, proposes three connector pattern to describe transient interactions in mobile computing that are inhibition, synchronization, and communication actions. Using the COMMUNITY design language for the syntax and the categorical framework Category Theory for the semantics, the authors illustrate how to obtain the architecture of a whole mobile system by applying the instantiated connectors to the components. This approach is suited for the representation of a static architecture of systems with limited mobility or limited number of different components types.
In [59] the authors describe a way to characterize connectors basing on the same foundational idea of Mendeleiev's periodic table of elements. Inspiring to this table that contains all basic elements in which all known substances can be decomposed, they propose a set of high level canonical properties as framework to describe all possible connectors and to allow operations to be defined over them. They also reason about factorization of connectors with common properties, specialization of connectors with the addition of properties to the owned set and highlight the class connector concept.

Summarizing, as explained in this section, our work clearly relates to mediator patterns existing in the literature in the software architecture area, in the software connector field, and especially to some design patterns and patterns in the web services context.

2.3 PROTOCOL INTEROPERABILITY

Protocol interoperability has been the focus of significant research since the early days of networking. Systematic approaches to **protocol conversion** were the initial focus of

studies, i.e., synthesizing a mediator that adapts the two interacting protocols that need to interoperate based on formal methods as surveyed in [29]. 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 [72] or correspond to the reference protocol associated with the service to be realized through protocol interaction [86]. 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 [29]. 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 top-down 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 [71]. Our work extensively builds on these formal foundations, adopting a bottom-up approach in the form of interface mapping which, under some assumptions, is always able to synthesize a mediator - if it exists. However, unlike the work of [72], 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).

A work strictly related to the theory presented in this thesis is the seminal work [130] that proposes an **adaptor theory** to characterize and solve the interoperability problem of augmented interfaces of applications. Yellin and Strom formally define the checks of applications compatibility and the concept of adapters. The latter can be used to bridge the differences discovered while checking the applications that have functional matching but are protocol incompatible. Furthermore, they provide a theory for the automated generation of adapters based on interface mapping rules, which relate to our definition of ontology mapping for protocols. With respect to the protocol conversion, in [130] is addressed a more practical treatment, 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 (as illustrated in Section 2.2). Then, our solution relates to this specific proposal and it is more general by dealing with more complex mediation patterns and further inferring message mapping from the ontology-based specification of interfaces. While the approach presented in [130] is semi-automatic because of non-automatic interface mapping. 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 [130] 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 recent works are for example [24, 23, 30, 117, 8, 7]. The paper [117] presents an approach for the synthesis of deadlock-free coordinators for correct components assembly. The aim of the coordinator is to guarantee that the assembly correctly interact by preventing the behavioural problems that can arise from component composition. The coordinator is build by taking as input the behavioural interfaces specification of components and the desirable behavioural properties for the composed system. The approach is supported by the SYNTHESIS tool[1] [7].
Moreover, in [8] the authors illustrate how to generate a distributed adaptor by splitting the centralized version that is the outcome of the approach in [117] into wrappers local to the components. Differently from our work, [117] is designed for protocols already able to interoperate, hence not dealing with the reasoning on different languages. Nevertheless, we can think to exploit some of their achievements in order to build a whole implementation of our process till the code generation.
Similarly to [117], the paper [30] proposes an approach for *software adaptation* which takes as input components behavioral interfaces and adaptation contracts and automatically builds an adaptor such that the composed system is deadlock-free. The adaptor generation is also tool supported.
Although the work described in [30] is a first step towards dealing with protocols on different languages, this remains at a syntactic level. Indeed, the author assume to know possible action mappings between an action of one protocol which is labelled with a different name in the other. Under this assumption also the approach [117] may be able to deal with such protocols.
Unfortunately, also [30] does not deal with the reasoning on different languages while building the interoperability solution and solving the behavioural mismatches.
The two approaches [117] and [30] seems to be closely related to our work while posing the focus on different problems. Indeed, they both mainly concentrate on ensuring deadlock freedom and on addressing system-wide adaptation, specified through policies and properties, while considering the communication problem (almost) solved. Instead our main goal, starting from a more complex scenario where protocols meet dynamically, is to achieve communication. We consider different protocols in terms of language and behaviour and without a priori knowledge one of the other. And our aim is to find - if possible - a way to let them communicate, i.e., we look for a common abstraction that allows us to synthesize a connector solving their mismatches and enabling their interoperability.

More recently, with the emergence of **Web services** and advocated universal interoperability, the research community has been studying solutions to the *automatic mediation of business processes* [119, 118, 80, 127]. They differ with respect to: (a) a priori exposure of the process models associated with the protocols that are executed by networked resources, (b) knowledge assumed about the protocols run by the interacting parties, (c) matching relationship that is enforced. However, most solutions are discussed informally,

[1] http://www.di.univaq.it/tivoli/index.php?pageId=synthesis

making it difficult to assess their respective advantages and drawbacks.

This highlights the needed for a new and formal foundation for mediating connectors from which protocol matching and associated mediation may be rigorously defined and assessed. These relationships should be automatically reasoned upon, thus paving the way for on the fly synthesis of mediating connectors. To the best of our knowledge, such an effort has not been addressed in the Web services and Semantic Web area although proposed algorithms for automated mediation manipulate formally grounded process models. Within the Web Services research community, a lot of work has been also devoted to *behavioral adaptation* which has been actively studying this problem. Among these works, and related to our, there is [81]. It proposes a *matching approach* based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioral descriptions. Our matching, as sketched before, is driven by the ontology and is better described in [110] where the theory underlying our approach is described at a high level and in [62] where a more detailed version of the theory can be found. Other related works are the ones about protocol conformance [21, 22].

Moreover, recently the Web services community has been also 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 [80, 97], latest work concentrates on automated, run-time solutions [40, 31]. The work [40] addresses the interoperability problem between services and provides experimentation on real Web2.0 social applications. They propose a technique to dynamically detect and fix interoperability problems based on a catalogue of inconsistencies and their respective adapters. This is similar to our proposal to use *ontology mapping* to discover mismatches and mediator to solve them. Our work differs with respect to theirs because we aim at automatically synthesizing the mediator. Instead, their approach is not fully automatic since although they discover and select mismatches dynamically, the identification of mismatches and of the opportune adapters is made by the engineer.
Our work also closely relates to [31], 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 [31]. However, our work goes one step forward by not being tied 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 [113]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [34, 119]. However, as for the above mentioned work on Web service adaptation, mediation is focused on the up-

per 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.

Other works concerns **connectors** and include a *classification framework* [78] presented with the purpose to better understand the existing connectors and also to synthesize new species. Assuming that connectors mediate the interactions among components, in order to give the classification they identify for each connector: one or more interaction channels called ducts, mechanisms for transferring data and/or control along the ducts, connector categories, types, dimensions within types, and values for each dimensions. The main categories identified are: communication, coordination, conversion, and facilitation while the main types are: procedure call, data access, linkage, stream, event, arbitrator adaptor, and distributor. Connector species are classified or created by choosing the appropriate combinations of dimensions and values for them.

Other *studies on connectors* are [128] and [69]. In [128] Woollard and Medvidovich propose software connectors as first-class to model services required in parallel programming instead of languages, libraries, and compilers. They both observed the general form with which scientific applications are described as programs and the evolution of connectors, from locks and semaphores to monitors and single commits. Thus basing on these observations they resumed the services that high performance connectors must provide that is communication, separation of concerns and synchronization.
In [69] Kell studies connectors under several perspectives. He says that a connector provides mechanisms enabling communication between components and requires agreements between two components which want to communicate in terms of common assumptions about syntax and semantics of the messages they exchange. He also states that there are types and instances of connector that can adhere or not to a type and that have in addiction run-time state. Moreover he discusses about the continuum that component and connector form (not a disjunction as is written till now in literature) and about the distinction that only depends on the level of abstraction assumed. Furthermore he describes the relationships with coordinators and adaptors that are viewed as most general connectors and argue about the coupling derived from the agreements giving three possible mitigation solutions that also achieve re-use: localization of the agreement to change it, standardization to eliminate coupling problems and adaptation with extra code to mediate. At the end he identifies the configuration languages highlighting their relevance in composition proposing explicit primitive connectors.

Other *formally grounded* state of the art works on connectors include [112] that presents an approach for formally specifying connector wrappers as protocol transformations, modularizing them, and reasoning about their properties, with the aim to resolve component mismatches. In their vision a wrapper is new code interposed between component interfaces and communication mechanisms and its intended effect is to moderate the be-

haviour of the component in a way that is largely transparent to the component or the interaction mechanism. Instead, a connector wrapper is a wrapper that address issues related to communication and compatibility including things such as changing the way data is represented during communication, the protocols of interaction, the number of parties that participate in the interaction, and the kind of communication support that is offered for things like monitoring, error handling, security, and so on. Their approach is to formally specify connector wrappers by means of a process algebra as a set of parallel process (one for each connector's interface and one for the glue) and to produce new connectors converting the protocol defining the first connector wrapper into a new protocol defining an altered connector by adding and modifying processes. Protocol transformations may include redirecting, recording and replaying, inserting, replacing, and discarding particular events yielding benefits like composability and reusability.
In [48] Fiadeiro, Lopes, and Wermelinger propose mathematical techniques as foundations to develop architectural design environments that are ADL-independent. In particular they present a categorical semantics for the formalization of concepts like interconnection, configuration, instantiation and composition. That is they show how this framework allows the computation (i.e. the configuration) of the program thus providing a well-defined semantics for an architectural graphical representation. They moreover show their formalization of connectors to coordinate the interactions (that can change at run-time) between components and how the categorical framework provides a way to create new connectors by composition.
In [11] Balek considering the context of distributed component-based applications and in particular the SOFA/DCUP project component model, proposes a new connector model that allows the description of more or less complex interactions between components with a semi-automatic generation of the corresponding code. He highlights the so called deployment anomaly that is the post-design modification of a component internals enforced by its deployment (owing to the fact that communication mechanisms are directly hard-coded in the component). Thus he identifies functional and non-functional requirements of a connector model design giving also an architectural model of connectors and a proof-of-concepts.
The paper [75] presents a formal specification mechanism, by a categorical semantics, for higher order connectors concept that is connectors that take a connector as parameter and deliver another as result. The connector in input is constituted by a formal parameter declaration that describes the kind of connector to which that service can be applied and a body connector that models the nature of the service that is superposed on instantiation of the formal parameter. This notion supports designers in constructing software connectors also including services for properties like security, fault tolerance, compression, and monitoring handled by the connector passed as actual parameter. They also describe how to compose these higher order connectors in a way independent of any Architectural Description Language.
In [12] Barbosa and Barbosa present a formalization of software connectors that is the patterns of interaction between components. They give a semantic model through a coalgebra and also a systematic way of building connectors (e.g. source and sinks connectors) by aggregation of ports or by a concurrent composition or by a generalization of pipelining (e.g. broadcasters and mergers).

In [27] Bruni, Lanese, and Montanari present an algebra for five basic stateless connectors that are symmetry, synchronization, mutual exclusion, hiding and inaction. They also give the operational, observational and denotational semantics and a complete normal-form axiomatization. The presented connectors can be composed in series and in parallel. In [104] Schreiner and Göschka propose the foundations for the automatic middleware generation following the component based paradigm principles applied to the communication middleware for distributed embedded systems. In particular, they provide the starting elements for the methodology that are: connector schematics i.e. structural design of explicit connectors (comprising an architecture of the communication needs) treating sender-receiver connector and client-server connector. They also provide a classification of a set of connectors building blocks called communication primitives.

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 reported in [36, 38], one-to-one bridging was among the early approaches [89] and then evolved into more generic solutions such as Enterprise Service Bus [32], interoperability platforms [56] and transparent interoperability platforms [25, 82]. 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 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 networking environment.

To summarize what we already mentioned above, our work: (i) builds on the protocol conversion idea of a bottom-up approach in the form of interface mapping; (ii) positions in the wide research area of software connectors and in particular of mediators; (iii) is linked to adaptors theories for several aspects, e.g., interface (ontology) mapping and mediation patterns, while is different for another aspect, i.e., our work concentrates on the problem of finding a way to let protocols communicate instead of ensuring deadlock freedom while assuming the communication (almost) solved; (iv) relates to and builds upon tremendous work done in the web services area going a step forward since the theory is not tight to a specific context; (v) relates to the middleware area contributing with a combined approach between application- and middleware-layer interoperability where middleware interoperability is among different middleware paradigms.

In conclusion, tremendous work has been done in the literature in several research areas as surveyed in this chapter. Although the existence of all these works, to the best of our

knowledge they have limitations making it difficult, if not impossible, to automatically synthesize mediators in the context we consider. We recall that we concentrate on a networked context including protocols that meet dynamically without any a-priori knowledge one another. This highlights the need for a reasoning on such protocols that let emerge a way for them to achieve communication (if possible) to interoperate.
As mentioned above, the solutions in the literature are limited for one or a combination of the following aspects.

- assume the communication problem solved (or almost solved) by considering protocols already (or almost) able to interoperate;
- are informal making it impossible to be automatically reasoned;
- deal with the interoperability separately either at application layer or at middleware layer;
- follows a semi-automatic process for the mediator synthesis requiring a human intervention;
- consider only few possible mismatches.

This motivates our work that aims at defining a process and a theory to automatically synthesize emerging mediators.

CHAPTER 3

THE AMAzING PROCESS

One of the contributions of this thesis work, as pointed out in Chapter 1, is *a comprehensive mediator synthesis process* to automatically synthesize a mediator that allows compatible protocols to interoperate.
Given the ubiquitous context where protocols meet dynamically without a priori knowledge, and since we focus on automated synthesis techniques, a key issue is to work with *reduced yet meaningful (protocols) models*. Moreover, if protocols we deal with are compatible, being able to potentially communicate, we expect to find at a given level of abstraction a *common (protocol) model* for them illustrating their potential interactions.

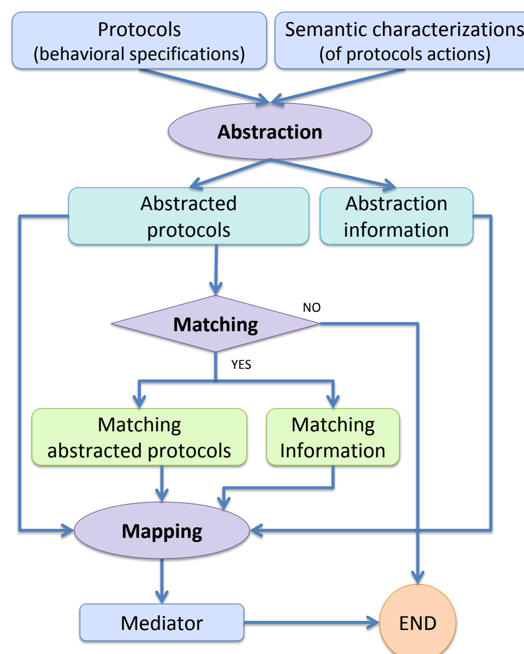


Figure 3.1: Overview of the AMAzING Process

These motivations inspired our process, depicted in Figure 3.1 which amounts to: abstracting the protocols so to make them comparable, while possibly reducing their size; checking the (abstracted) protocols compatibility, i.e., the existence of portions of them that could interoperate; deriving the mediator that, solving possible mismatches, allows

these protocols (portions) to communicate. We called this process AMAzING.
The acronym AMAzING summarizes the three phases or steps of the process that are: Abstraction, MAtching and mappING respectively illustrated in Sections 3.1, 3.2, and 3.3. Chapters 5 and 6 illustrate a realization the AMAzING process from abstraction to matching and then mapping. In the following sections, for the explanation, we refer to the *Photo Sharing scenario*. We consider two applications dynamically meeting. A spectator enters the stadium with its device embedding a peer-to-peer (P2P) photo consumer application; another spectator is already inside with an infrastructure based (IB) producer application running on its device.

3.1 THE ABSTRACTION PHASE

The first phase of the AMAzING process is the *abstraction* which is highlighted by Figure 3.2 by the (non-gray) coloured portion. Referring to the scenario described above, although the two application protocols are heterogeneous in principle they should be compatible and able to communicate. In order to check their compatibility, during the matching phase, we first need to make them comparable.

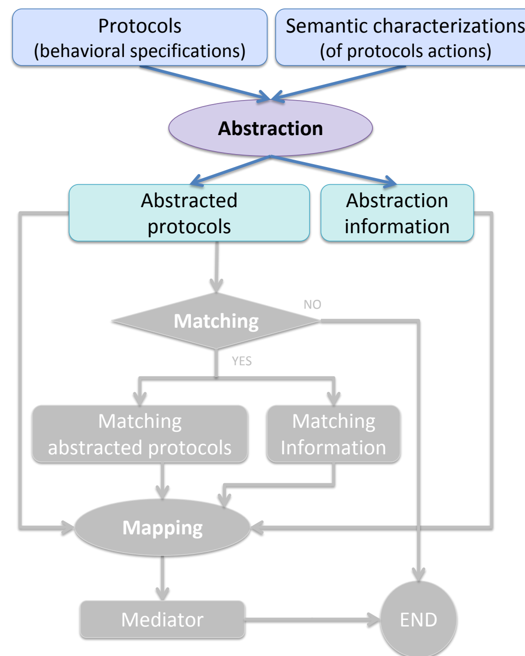


Figure 3.2: Abstraction phase of the AMAzING Process

Then, this step takes as input (i) protocols behavioural models, and (ii) semantic characterization of the protocol actions. The returned output is made by (1) abstracted protocols behaviour and (2) information used to perform the protocol abstraction. This abstraction information, in case of a positive answer from the subsequent matching phase, will be exploited (backwards) to build the mediator during the mapping phase. In order to find

the protocols' abstractions, we exploit the information contained in their semantic characterization to suitably relabel them so to make them comparable.
In the example described above we have two different protocols (i), P2P consumer and IB producer, having different languages, interactions with third parties, and different granularities. However, thanks to their semantic characterization (ii), and in particular to some semantic matching information (2), we are able to abstract their behaviours by relabelling. After the relabelling operation, we obtain new behavioural descriptions for both protocols labelled only by common actions and τ, that is, protocols more abstract than before, e.g., sequences of actions may have been compressed into single actions. This step, producing abstract protocols, addresses to some extent a scalability issue. The next phase is the matching.

3.2 The Matching Phase

To establish whether the two abstracted protocols can interoperate we have to check the existence of portions that can interoperate.

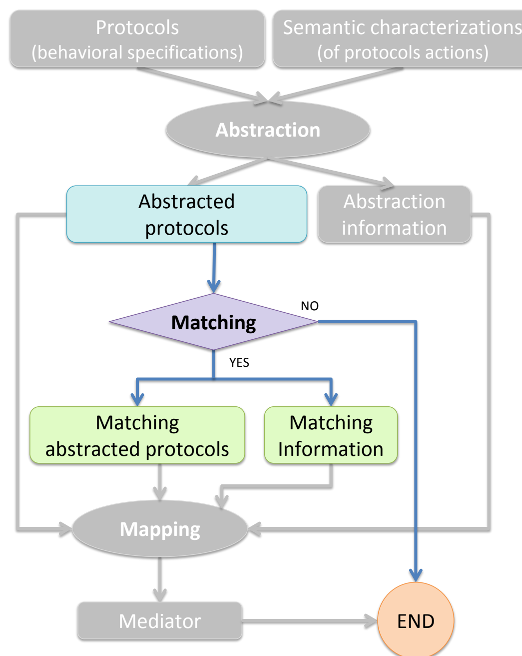


Figure 3.3: Matching phase of the AMAzING Process

More precisely we have to verify the existence of at least one complementary trace, by properly taking into account and managing mismatches (see Chapter 4) and communications with third parties. Figure 3.3 illustrates the second phase of the AMAzING process, the *matching*. The non-grayed portion shows that this step takes as input (1) the abstracted behavioural specification and, performs the matching check that can answer yes

or no. Depending on the answer, this phase can return as output (a) the matching abstracted behaviours and (b) matching information (case answers yes) or end its execution (case answer no). In order to find the matching portion of the abstracted protocols, if it exists, a check has to be performed to discover whether modulo mismatches and third parties communication there exists at least one trace in both protocols that allows them to coordinate to reach their goal (we recall that we express coordination as synchronization). If such trace(s) exists then the check is successful, otherwise it is not.
The matching check of the above Photo Sharing example is successful and the matching abstracted behaviours of IB producer and P2P consumer are respectively upload/-download of photos and download/upload of comments. In this example we do not have matching information while in general they could be for instance actions reordering.
A successful matching means that the two systems can achieve a common goal/intent, i.e., implies the existence of a mediating connector and the automated computation of the mapping. After a successful matching check on the behaviours, we obtain the set of their complementary traces (labelled only by common actions and τ). The subsequent step is the mapping or synthesis which automatically produces the mediator protocol.

3.3 The Mapping Phase

The *mapping* or synthesis is the last phase of the process.

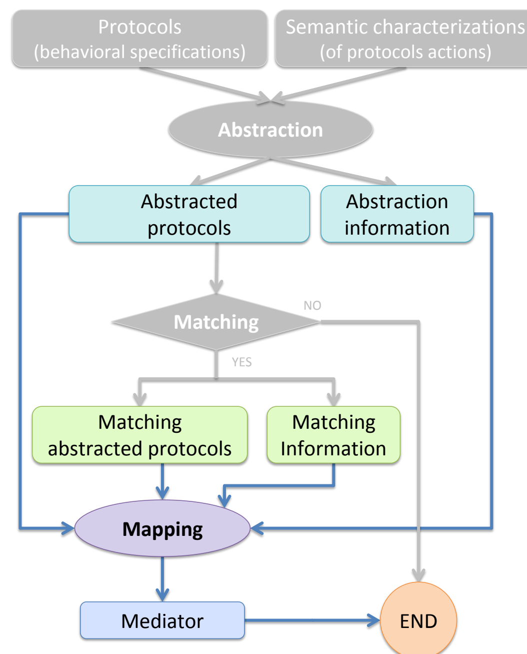


Figure 3.4: Mapping phase of the AMAzING Process

This step serves to find a suitable mediator that allows the protocol interoperability that otherwise would not be possible. It is automatically activated only in case of a success-

ful matching check and it is represented in Figure 3.4 by the non-gray coloured portion. he mapping takes as input the output of the two previous phases, i.e., (1) the abstracted protocols, (2) the abstraction information, (a) the matching abstracted behaviours, (b) and the matching information and returns as output a mediator. The mediator is described at the same level of the two starting protocols being labelled with actions belonging to their alphabets.
In the considered Photo Sharing the mediator allows: the IB producer to upload photos and the P2P consumer to download photos following their protocols while managing their previously identified mismatches, i.e., language differences, third parties interactions and different granularity. The mediator similarly allows the P2P consumer to upload photo comments and the IB producer to download them.
After the mapping, we then obtain a mediator that, as described in Chapter 1, is a protocol that allows the communication among the mismatching protocols by mediating their differences.

CHAPTER 4

MEDIATOR PATTERNS

In order to comprehensively define the problem of protocol mediation, we categorize the types of protocol mismatches that may occur and that must be solved in order to provide corresponding solutions to these recurring problems. This immediately reminds of patterns [5, 28, 9, 54]. Indeed, in this chapter, which is an extended and revised version of the work discussed in [109], we describe the *Mediator Patterns*, a set of design building blocks to tackle in a systematic way the protocol mediation problem, that is, the interoperability between heterogeneous protocols. The patterns give an overview about the kind of problems and their related solutions that have to be supported, and thus are accounted for in the definition of *functional matching* and *protocol mapping* used in the second and third steps of the AMAzING process.

The design building blocks that we present include:

- An Architectural Pattern called *Mediating Connector*, which is the key enabler for communication;
- A set of *Basic Mediator Patterns*, which describe: (i) basic mismatches that can occur while components try to interact, and (ii) their corresponding solutions.

For illustration, in the following, we consider the Photo Sharing scenario introduced in Section 1.1, for which multiple versions of both the IB and P2P implementations may be envisioned.
These applications should be able to interoperate since they both have similar functionalities while not having the very same interfaces and behaviours. Then, mediating their respective protocols to achieve interoperability is far from trivial.

In this chapter, we make some assumptions. We assume to know the interaction protocols run by two networked components as LTSs and the components' interfaces with which to interact as advertised or as result of learning techniques [60, 19]. We also assume a semantic correspondence between the messages exchanged among components exploiting ontologies. Note that we use the generic term component to indicate networked systems. In particular, to ease the explanation and without loss of generality, we consider one application at a time on networked systems.

The remaining of this chapter is organized as follows. In Section 4.1, we illustrate models of different implementations of the Photo Sharing scenario. In Section 4.2, we describe a pattern-based approach which we envision for the automatic synthesis of mediating connectors for the ubiquitous networked environment. In Sections 4.3 and 4.4, we illustrate the *Mediating Connector Architectural Pattern* and the related *Basic Mediator Patterns* Then, Section 4.5 illustrates the application of the mediator patterns to the Photo Sharing scenario (see Section 1.1) by showing how the defined patterns can be used to solve interoperability mismatches. In Section 4.6, we conclude by also outlining future work.

4.1 PHOTO SHARING APPLICATIONS

To better illustrate protocol mediation and the related underlying problems, in the following we exploit the Photo Sharing scenario (sketched in Section 1.1 and recalled above), where different application implementations with similar functionalities should be able to interoperate despite behavioural mismatches.
We recall that the high level functionalities that the networked systems implement, taking the producer perspective, are: (1) the *authentication* -for the IB producer only- possibly followed by (2) the *upload of photo*, by sending both metadata and file, possibly followed by (3) the *download of comments*; on the other hand, taking the consumer perspective, the implemented high level functionalities are: (i) the *download of photo* by receiving both metadata and file respectively, possibly followed by (ii) the *upload of comments*.
Figure 4.1 then shows four different versions of the P2P application (v1, v2, v3 and v4 respectively), where protocols are depicted using state machines (made by states, transitions and actions) where the names of actions are self-explanatory. Note that the four protocols differ for several aspects. For instance they have: different actions names with the same semantics (e.g., PictureFile vs. PhotoPhile), different actions order to perform the same operation (e.g., PhotoMetadata,PhotoFile vs. PhotoFile,PhotoMetadata), different actions granularity (e.g., PhotoMetadata and PhotoFile vs. PhotoMetaAndFile). We further use the convention that actions with overbar denote output actions while the ones with no overbar denote input actions. In all four versions of the P2P implementation, the networked system implements both roles of *producer* and *consumer*. Instead, as depicted in Figure 4.2, the IB implementation, while having similar roles and high level functionalities with respect to the P2P one, differs from it, because: (i) in IB, the *consumer* and *producer* roles are played by two different/separate networked systems, in collaboration with the server, and (ii) comparing complementary roles among any P2P and IB, they have different interfaces and behaviours. This second difference applies also if one considers two different versions (among the four) of the P2P implementation. In fact, two instances of different versions (e.g., Photo Sharing version 1 and version 3) have different interfaces and behaviours preventing their direct interoperability. While two instances of the same version (e.g., P2P Photo Sharing version 1) have the very same interfaces and behaviours thus being able to interoperate.

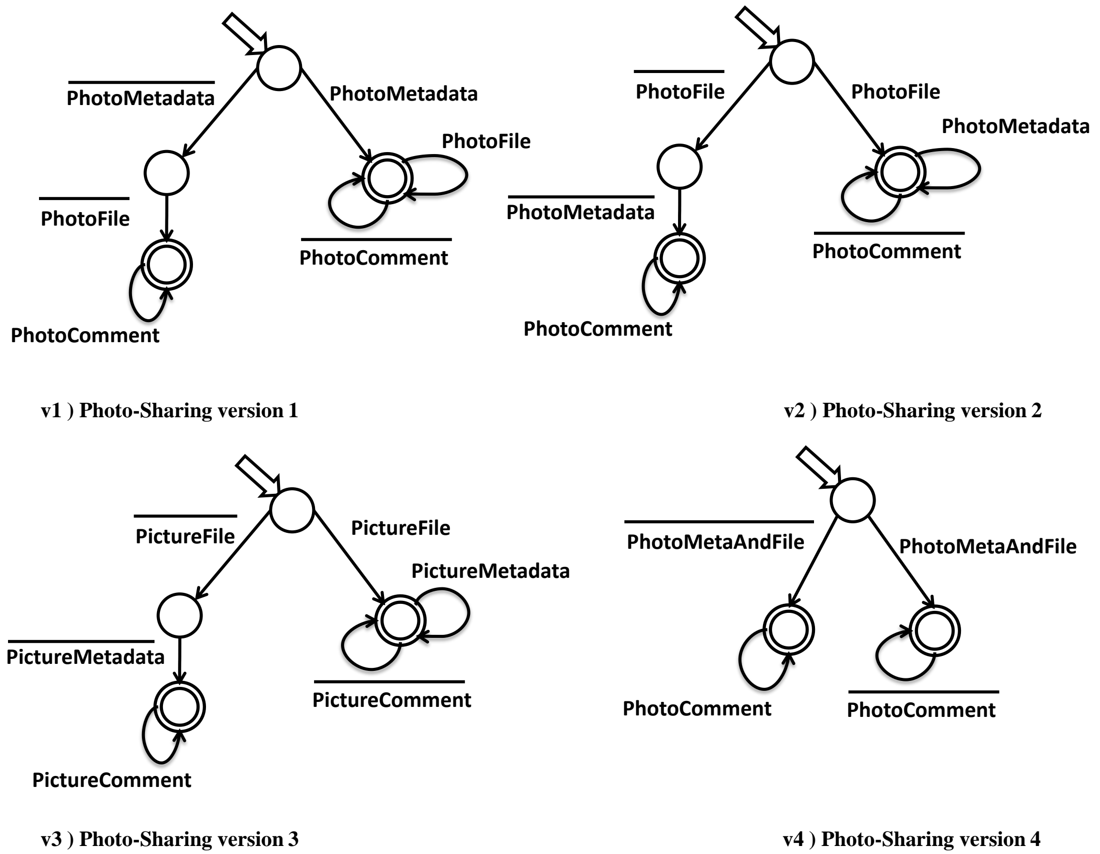


Figure 4.1: Peer-to-Peer-based implementation

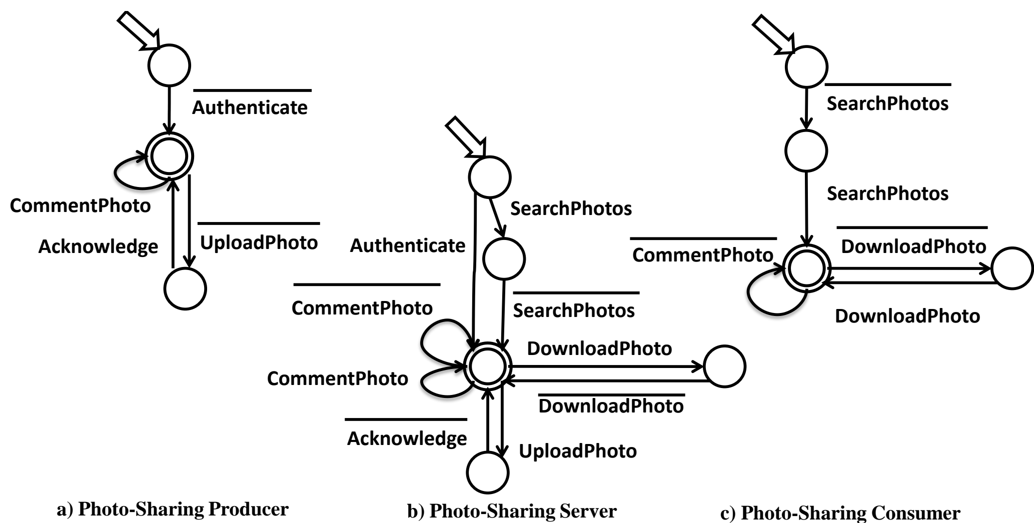


Figure 4.2: Infrastructure-based implementation

For the sake of illustration, from now on and when not differently specified, we consider as example the pair of mismatching applications made by: the IB producer (Figure 4.2 a)) and the P2P Photo Sharing Version 1 (Figure 4.1 v1)).

4.2 A Pattern-based Approach for Interoperability Mismatches

In this section we illustrate a pattern-based approach to solve interoperability mismatches. The defined patterns allow us to rigorously characterize the kind of interoperability mismatches we want to solve by means of the automated mediator synthesis process described in Chapters 3 and 5. Thus, this chapter can be considered as the foundational base for the subsequent chapters. To establish if components are compatible, i.e., if modulo some adaptation, they show complementary sequences of messages visible at interface level, we envision (1) a decomposition strategy/tool to decompose the whole components' behaviour (LTS) into *elementary behaviours* (traces) representing *elementary intents* of the components and (2) an automatic analyzer to identify mismatches between elementary behaviours of the different components as done in other research areas [81, 84]. Once discovered the components compatibility, solving their interoperability means solving the behavioural mismatches that they exhibit. Then it is necessary to: (3) define a mismatches manager to solve the identified mismatches between elementary behaviours; (4) define a composition approach to build elementary mediating behaviours (mediating traces) based on the identified mismatches and their relative solutions; (5) define a composition strategy to build a mediating connector's behaviour starting from the elementary mediating behaviours.

The above described approach is far from trivial, especially if it has to be automatically performed. However, in the following we contribute to its realization by describing six Basic Mediator Patterns that are the building blocks on which the steps (2), (3), and (4) can be built upon.

4.3 Mediating Connector Architectural Pattern

The interoperability problem between diverse components populating the ubiquitous environment and its related solution is characterized as a *Mediating Connector Architectural Pattern* basing on the template used in [28] that contains the following fields: Name, Also Known As, Example, Context, Problem, Solution, Structure, Dynamics, Implementation, Example Resolved, Variants, Consequences. The Mediating Connector is a behavioural pattern and represents the architectural building block embedding the necessary support to dynamically cope with components' behavioural diversity.

Name. Mediating Connector.

Also Known As. Mediator.

Example. As example we consider the Photo Sharing applications and in particular the infrastructure-based photo producer of Figure 4.2 a) and peer-to-peer Photo Sharing version 1 of Figure 4.1 v1).
Other example could also be, for instance, the infrastructure-based producer (Figure 4.2 a)), may want to communicate with the peer-to-peer Photo Sharing version 1 (Figure 4.1 v1)) (or also with version 2, 3, or 4 - Figure 4.1 v2), Figure 4.1 v3) or Figure 4.1 v4) respectively), and in principle this should be possible. Nevertheless, their behavioural mismatches prevent the communication.

Context. The environment is distributed and changes continuously. Heterogeneous (mismatching) systems populating the environment require seamless coordination and communication.

Problem. In order to support existing and future systems' interoperability, some means of mediation is required. From the components' perspective, there should be no difference whether interacting with a peer component, i.e, using the very same interaction protocol, or interacting through a mediator with another component that uses a different interaction protocol. The component should not need to know anything about the protocol of the other one while continuing to "speak" its own protocol.

Using the Mediating Connector, the following *forces* (aspects of the problem that should be considered when solving it [28]) need to be balanced: (a) the different components should continue to use their own interaction protocols. That is components should interact as if the Mediating Connector were transparent; (b) the following *basic interaction protocol mismatches* should be solved in order for a set of components to coordinate and communicate (a detailed description of these mismatches is given within Section *Basic Mediator Patterns*): 1) Extra Send/Missing Receive Mismatch; 2) Missing Send/Extra Receive Mismatch; 3) Signature Mismatch; 4) Ordering Mismatch; 5) One Send-Many Receive/Many Receive-One Send Mismatch; 6) Many Send-One Receive/One Receive-Many Send Mismatch.

Solution. The introduction of a Mediating Connector to manage the interaction behavioural differences between compatible components. The idea behind this pattern is that, by using the Mediating Connector, components that would need some interaction protocol's adaptation to become compatible, and hence to interoperate, are able to coordinate and communicate achieving their goals/intents without undergoing any modification.

The Mediating Connector is one (or a set of) component(s) that manages the behavioural mismatches listed above. It directly communicates with each component by using the component's proper protocol. The mediator forwards the interaction messages from one component to the other by making suitable translation/adaptation of protocols when/if needed.

Structure. The Mediating Connector Pattern comprises three types of participating components: communicating components, compatible components and mediators.
The *communicating components (or applications)* implement components able to directly interact with other components and evolve following their usual interaction behaviour. The *compatible components (or applications)* implement the application level entities (whose behaviour, interfaces' description and semantic correspondences are known). Each component wants to reach its intents by interacting with other components able to satisfy its needs, i.e. required/provided functionalities. However the components are unable to directly interact because of protocol mismatches. Thus, the compatible components can only evolve following their usual interaction behaviour, without any change. The *mediators* are entities responsible for the mediated communication between the components. This means that the role of the mediator is to make compatible components that are mismatching. That is, a mediator must receive and properly forward requests and responses between compatible components that want to interoperate. Figure 4.3 shows the objects involved in a mediated system first without and then with Mediating Connector.

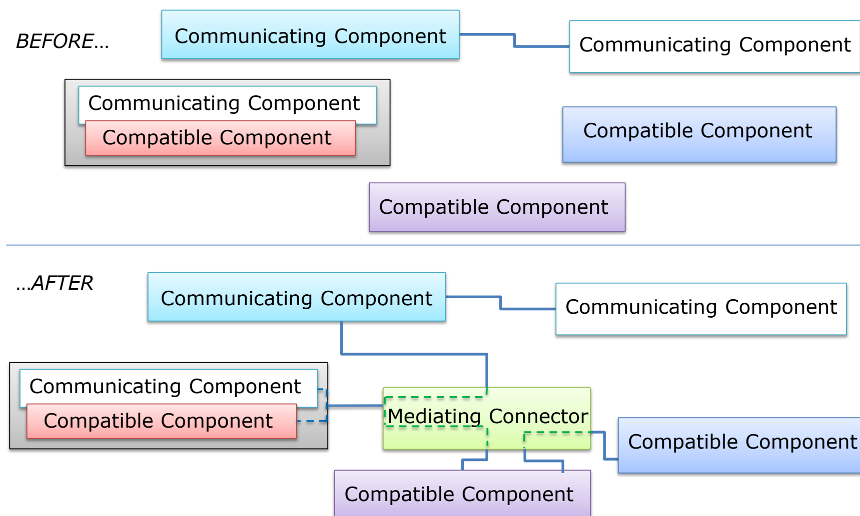


Figure 4.3: Entities involved in a mediated system without and with Mediating Connector

Dynamics. The dynamics refers to the interactions between components (applications) and mediators. In our case study, the IB producer protocol (Figure 4.2 a) performs one of its possible behaviours: it authenticates and then uploads one photo receiving the respective acknowledgment. This is performed by sending in sequence the messages *Authenticate* and *UploadPhoto* and then receiving the message *Acknowledgment*. The mediator should: (1) forward the authentication message as it is between the IB producer and its authentication server, that are communicating components, (2) manipulate/translate and forward the upload and acknowledge messages between the IB producer (Figure 4.2 a) and the P2P Photo Sharing version 1 protocol (Figure 4.2 v1)) that are compatible components. With the term "translation" we mean not just a language translation but also a "behavioural translation" (see Section *Basic Mediator Patterns* for details).

Implementation. The implementation of this pattern implies the definition of an ap-

proach/tool (we have sketched one in Section 4.2) to automatically synthesize the behaviour of the Mediating Connector which allows the compatible components to interoperate mediating their interactions.

Example Resolved. The Mediating Connector's concrete protocol for the example is shown in Figure 4.10. Once established that components are compatible (i.e. they have some complementary portion of interaction protocols), the mediating connector manages the components' behavioural mismatches allowing them to have a mediated coordination and communication.

Variants. Distributed Mediating Connector. It is possible to implement this pattern either as a centralized component or as distributed components, that is by a number of smaller components. This introduces a synchronization issue that has to be taken into consideration while building the mediator behaviour.

Consequences. The main *benefit* of the Mediating Connector Pattern is that it allows interoperability between components that otherwise would not be able to do it because of their behavioural differences. These components do not use the very same observable protocols and this prevents their cooperation while, implementing similar functionalities, they should be able to interact. The main *liability* that the Mediating Connector Pattern imposes is that systems using it are slower than the ones able to directly interact because of the indirection layer that the Mediating Connector Pattern introduces. However the severity of this drawback is mitigated and made acceptable by the fact that such systems, without mediator, will not be not able at all to interoperate.

4.4 Basic Mediator Patterns

In the previous sections we characterized the Mediating Connector pattern and we sketched an approach for the automatic synthesis of its behaviour.

In this section, we concentrate on six finer grain *Basic Mediator Patterns* which represent a systematic approach to solve interoperability mismatches that can occur during components' interaction. The Basic Mediator Patterns are constituted by basic interoperability mismatches with their corresponding solutions and are: (1) Message Consumer Pattern, (2) Message Producer Pattern, (3) Message Translator Pattern, (4) Messages Ordering Pattern, (5) Message Splitting Pattern, (6) Messages Merger Pattern.

The mismatches, inspired by service composition mismatches, represent send/receive problems that can occur while synchronizing two traces. We are not considering parameters mismatches which are extensively addressed elsewhere [85].

Figure 4.4 shows the basic interoperability mismatches that we explain in detail in the following. For each basic interoperability mismatch, we consider two traces (left and right) coming from two compatible components. All the considered traces are the most elementary with respect to the messages exchanged and only visible messages are shown.

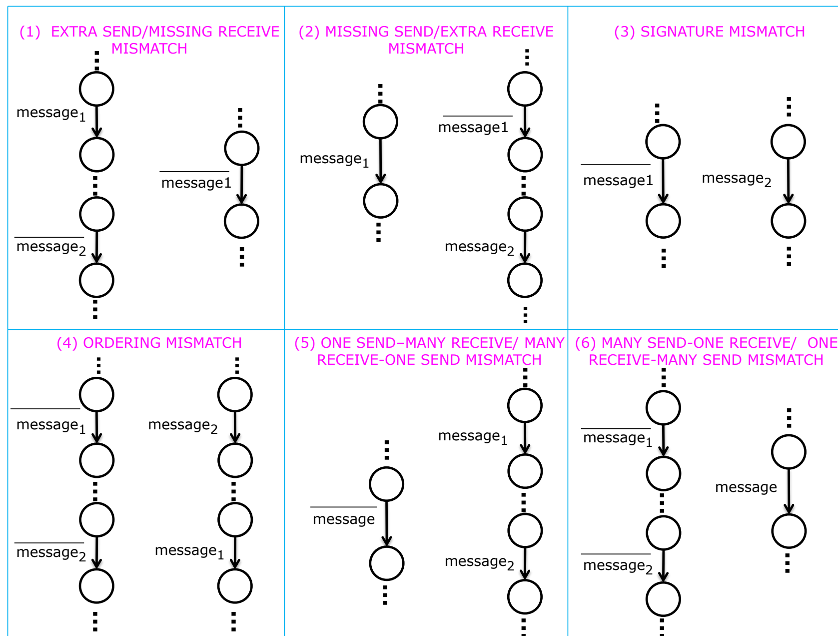


Figure 4.4: Basic interoperability mismatches

It is obvious that, in real cases, the traces may also contain portions of behaviour already compatible (abstracted by dots in the figure) and may amount to any combination of the presented mismatches. Then an appropriate strategy to detect and manage this is needed. The considered basic mismatches are addressed by the basic solutions (elementary mediating behaviours) illustrated in Figure 4.5 where only their visible messages are shown (messages that they exchange with the components).

The six Basic Mediator Patterns share the context, i.e., the situation in which they may apply and have a unique intent.

Context. Consider two traces (left and right) expressing similar complementary functionalities. Focus on one of their subtraces which identifies an elementary action that is semantically equivalent and complementary between them.

Intent. To allow synchronization between the two traces letting them evolve together. This, otherwise would not be possible because of behavioural mismatches.

(1) MESSAGE CONSUMER PATTERN.

Problem. (1) Extra send/missing receive mismatch ((1) in Figure 4.4, where the extra send action is $\overline{message_2}$). One of the two considered traces either sends an extra action or a it lacks a receive action.

Example. Consider two traces implementing the abstract action “upload photo (respectively download photo)”. For example, in the mismatch (1) of Figure 4.4 the right trace implements only the sending of the photo ($\overline{message_1}$) while the left trace implements the

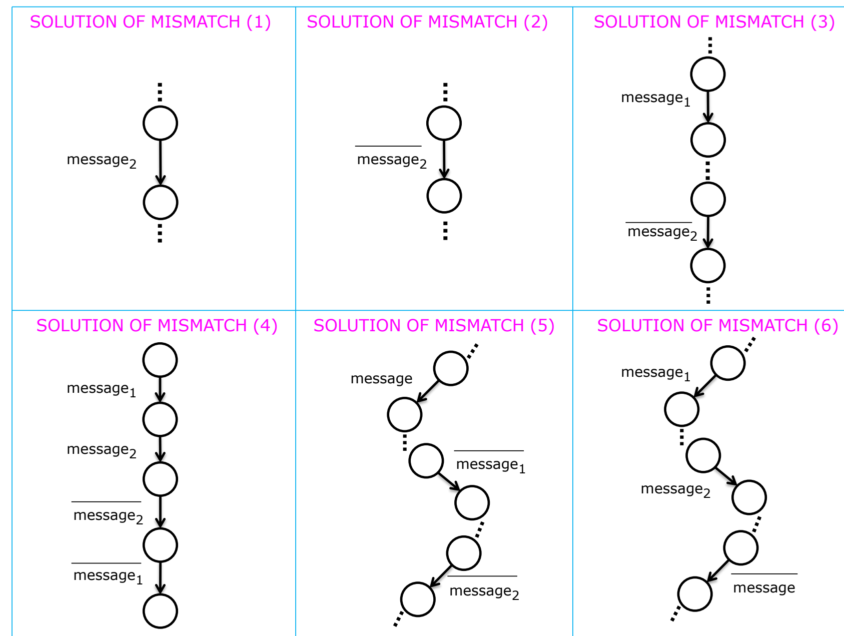


Figure 4.5: Basic solutions for the basic mismatches

receiving of the photo and the sending of an acknowledgment ($message_1.\overline{message_2}$).

Solution. Introducing a *message consumer* (solution of mismatch (1) in Figure 4.5) that is made by an action that, "consumes" the identified extra send action by synchronizing with it, letting the two traces communicate.

Example Resolved. First the two traces synchronize on the sending/receiving of the photo ($message_1$) and then the left trace synchronizes its sending of the acknowledgment ($\overline{message_2}$) with the message consumer that receives it.

Variants. Possible variants and respective solutions are represented in Figure 4.6 and are:

(a) $message_1$ has switched the send/receive type within the two traces, i.e., the left trace is the sequence $\overline{message_1}.\overline{message_2}$ while the right trace is just $message_1$. In this case the message consumer remains the same ($message_2$).

(b) $\overline{message_1}$ is the extra send message instead of $\overline{message_2}$. The left trace is the sequence $\overline{message_1}$. $\overline{message_2}$ while the right trace is made by $message_2$. In this case the message consumer performs $message_1$ absorbing the extra sent message.

(c) the extra send message is $\overline{message_1}$, the left trace is the sequence $\overline{message_1}$. $message_2$ while the right trace is $\overline{message_2}$. In this case the message consumer is made by $message_1$.

(2) MESSAGE PRODUCER PATTERN.

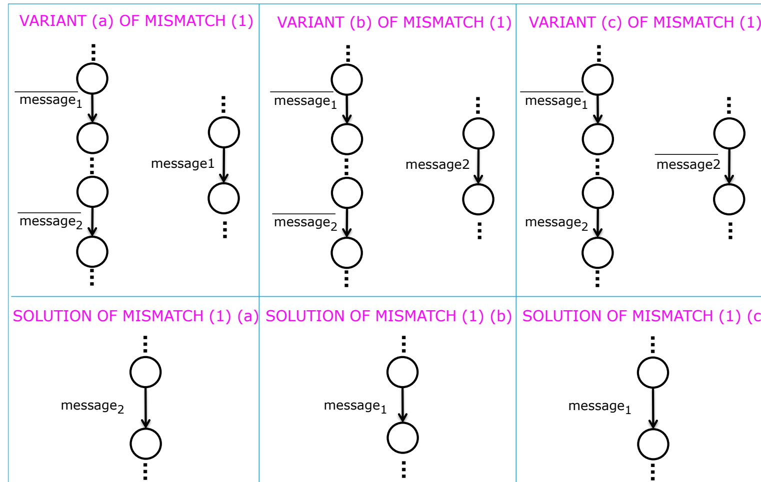


Figure 4.6: Variants of the Basic Mediator Pattern (1)

Problem. (2) Missing send/extra receive mismatch ((2) in Figure 4.4, where the missing send action is $\overline{message_2}$). One of the two considered traces either contains an extra receive action or a send action is missing in it. This is the dual problem of mismatch (1).

Example. Consider two traces implementing the abstract action "send (respectively receive) photo". In the mismatch (2) of Figure 4.4, the right trace implements the sending of the photo ($\overline{message_1}$) and the receiving of an acknowledgment ($message_2$) while the left trace implements just the receiving the message ($message_1$).

Solution. Introducing a *message producer* (solution of mismatch (2) in Figure 4.5) made by an action that "produces" the missing send action corresponding to the identified extra receive action and let the two traces synchronize.

Example Resolved. The two traces first synchronize on the sending/receiving of the message ($message_1$) and then the right trace synchronize its receive of the acknowledgment ($message_2$) with the message consumer mediator that sends it.

Variants. Possible variants and respective solutions are shown in Figure 4.7 and are:

(a) $message_1$ has switched the send/receive type within the two traces, i.e., the left trace is $\overline{message_1}$ while the right trace is the sequence $message_1.message_2$ In this case the message producer performs $\overline{message_2}$.

(b) the missing send message is $\overline{message_1}$, instead of being $message_2$, the right trace is the sequence $message_1.\overline{message_2}$ while the left trace is made by $message_2$. In this case the message producer is made by $\overline{message_1}$.

(c) the missing send message is $message_1$, the left trace is $message_2$ while the right trace is the sequence $message_1.\overline{message_2}$. In this case the message producer is made by $\overline{message_1}$.

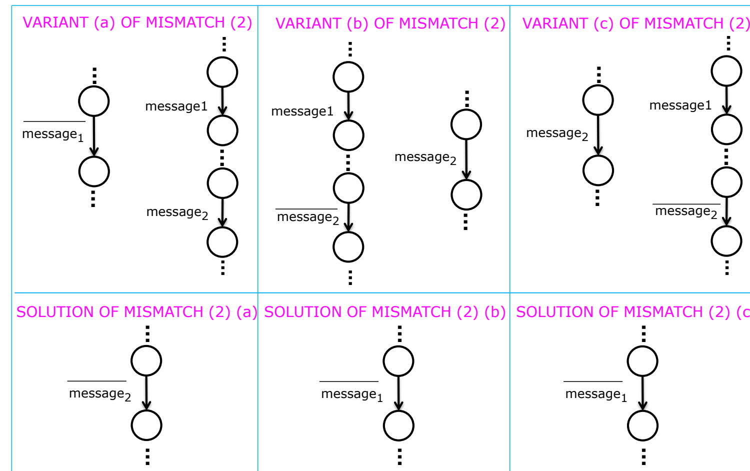


Figure 4.7: Variants of the Basic Mediator Pattern (2)

(3) MESSAGE TRANSLATOR PATTERN.

Problem. (3) Signature mismatch (upper right box of Figure 4.4). The two traces represent semantically complementary actions but with different signatures. With signature we mean only the action name.

Example. Consider two traces implementing the abstract action "send (respectively receive) photo file". Instantiating the mismatch (3) of Figure 4.4, $\overline{message_1}$ could be the send of a message *PhotoFile* while $message_2$ the receive of a *PictureFile* message.

Solution. Introducing a *message translator* (solution of mismatch (3) in Figure 4.5). It receives the request and sends it after a proper translation[1]. Referring to the example, the translator mediator trace is: $message_1.\overline{message_2}$.

Example Resolved. First the message *PhotoFile* is exchanged between one trace and the mediator. After its translation, a *PictureFile* message is sent by the mediator to the other trace. The message translator performs: *PhotoFile*.$\overline{PictureFile}$.

Variants. A possible variant with its solution is shown in Figure 4.8 and amounts to exchanging the sender/receiver roles between the two traces, i.e., $message_1$ and $\overline{message_2}$ and the solution is made by $message_2$. $\overline{message_1}$.

(4) MESSAGES ORDERING PATTERN.

Problem. (4) Ordering mismatch ((4) in Figure 4.4, where both traces perform complementary (send/receive) $message_1$ and $message_2$ but in different order). Both traces consist of complementary functionalities but they perform the actions in different orders. Nevertheless this mismatch can be considered also as a combination of extra/missing

[1]Technically the message translator synchronizes twice with the involved components using different messages and this implements a translation.

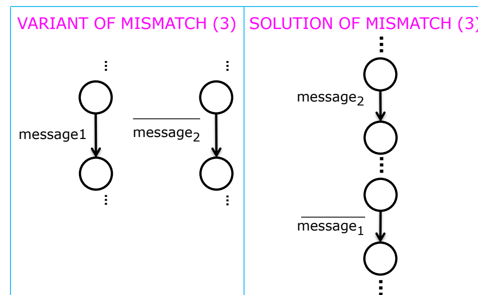


Figure 4.8: Variants of the Basic Mediator Pattern (3)

send/receive actions mismatches (1) and (2), however we choose to consider it as a first class mismatch. Generally speaking, it may happen that not all the ordering problems are solvable due to the infinite length of the traces. However this is not our case because we work on finite traces that represent a conversation.

Example. Consider two traces implementing the abstract action "send (respectively receive) photo". $message_1$ and $message_2$ in the mismatch (4) of Figure 4.4, for example, correspond to $PhotoMetadata$ and $PhotoFile$ respectively. Then, one sends the sequence $PhotoMetadata \,.\, PhotoFile$ while the other receives $PhotoFile \,.\, PhotoMetadata$.

Solution. Introducing a *messages ordering* (solution of mismatch (4) in Figure 4.5). This pattern has a compatible behaviour for both the traces. The pattern is made by a trace that receives the messages and, after a proper reordering, resends them.

Example Resolved. Referring to the example, the messages ordering trace is: $message_1 \,.\, message_2 \,.\, \overline{message_2} \,.\, \overline{message_1}$ that is $PhotoMetadata \,.\, PhotoFile \,.\, \overline{PhotoFile} \,.\, \overline{PhotoMetadata}$. That is, first one trace synchronizes with the mediator which receives the messages and then the mediator reorders the messages and sends them to the other trace.

Variants. Possible variantsand respective solutions are shown in Figure 4.9 and are:

(a) left trace has switched the sender/receiver role with respect to the right trace, i.e., the left trace is the sequence $\overline{message_2}.\overline{message_1}$ while the right trace is the sequence $message_1.message_2$. In this case the messages ordering is the sequence $message_2.message_1.\overline{message_1}.\overline{message_2}$.

(b) in both traces the first action is a send while the second is a receive. That is, the left trace is $\overline{message1}.message2$ while the right is $\overline{message2}.message1$. In this case the messages ordering is the sequence $message_1 \,.\, message_2 \,.\, \overline{message_2} \,.\, \overline{message_1}$.

(c) in both traces the first action is the receive followed by the send. That is, the left trace is $message_1 \,.\, \overline{message_2}$ while the right is $message_2.\overline{message_1}$. In this case the basic solution to solve the mismatch is not the messages ordering. It is a proper

combination of messages producers and consumers (message producer followed by message consumer for the left trace followed by message producer followed by message consumer for the right trace). That is, $\overline{message_1}.message_2$ followed by $\overline{message_2}.\ message_1$.

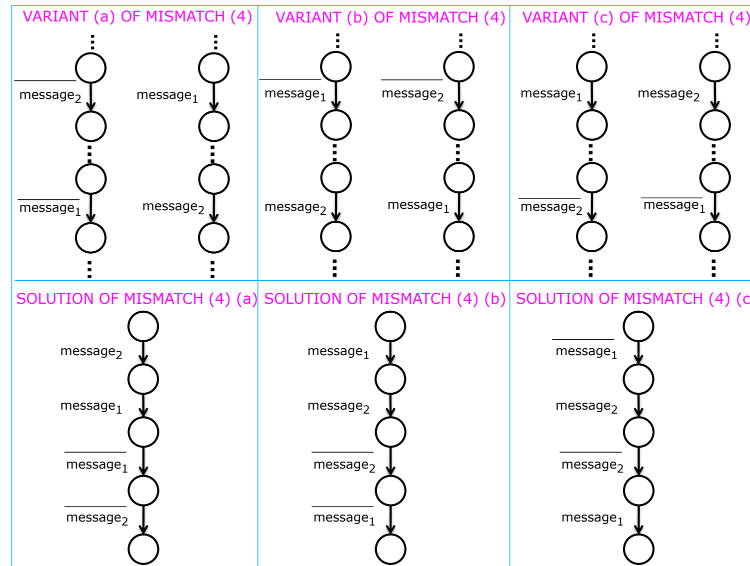


Figure 4.9: Variants of the Basic Mediator Pattern (4)

(5) MESSAGE SPLITTING PATTERN.

Problem. (5) One send-many receive/many receive-one send mismatch ((5) in Figure 4.4). The two considered traces represent a semantically complementary functionality but one expresses it with one action and the other with two actions.

Example. Consider two traces implementing the abstract action “send (respectively receive) photo”. Instantiating the one send-many receive mismatch (5) of Figure 4.4, for example, $\overline{message}$ can be the send of one message $\overline{PhotoMetaAndFile}$ while $message_1$ and $message_2$ are the receive of two separate messages $PhotoMetadata$ and $PhotoFile$.

Solution. Introducing a *message splitting* (solution of mismatch (5) in Figure 4.5). It receives one message from one side, splits it properly, and sends the split messages to the other[2]. Referring to the example, the trace of the message splitting (5) is: $message$. $\overline{message_1}$. $\overline{message_2}$.

Example Resolved. With respect to the example, the mediator first performs one receive, then a splitting, and subsequently sends two messages. That is, of $PhotoMetaAndFile$. $\overline{PhotoMetadata}$. $\overline{PhotoFile}$.

(6) MESSAGES MERGER PATTERN.

[2]Technically the message splitting synchronizes several times with the involved components using different messages and this implements a split.

Problem. (6) Many send-one receive/one receive-many send mismatch ((6) in Figure 4.4). The two considered traces represent a semantically complementary functionality but they express it with a different number of actions. This is the dual problem of mismatch (5).

Example. Consider two traces implementing the abstract action "send (respectively receive) photo". Instantiating the many send-one receive (6) of Figure 4.4, for example, $\overline{message_1}$ and $\overline{message_2}$ are the sending of two separate messages $\overline{PhotoMetadata}$ and $\overline{PhotoFile}$ while $message$ is the receiving of $PhotoMetaAndFile$.

Solution. Introducing a *messages merging* (solution of mismatch (6) in Figure 4.5). It receives two messages from one side, merges them properly, and sends the merged messages to the other. Referring to the example, the trace of the messages merging is: $message_1.message_2.\overline{message}$.

Example Resolved. With respect to the example, the mediator first performs two receives, then a merge, and subsequently sends one message. That is, $PhotoMetadata$. $PhotoFile$. $\overline{PhotoMetaAndFile}$.

4.5 Application of the Patterns to the Photo Sharing Scenario

The aim of this section is to show the patterns at work, putting together all the jigsaws puzzle. Thanks to a *compatibility analyzer* (e.g., see the definition of functional matching in the two next chapters), we discover that the two Photo Sharing applications considered as example (i.e., the IB photo producer (Figure 4.2 a)) and the P2P Photo Sharing version 1 (Figure 4.1 v1))) are compatible, since they share some intent, having complementary portions of interaction protocols. Hence, it makes sense to use the architectural Mediating Connector Pattern to mediate their conversations.

Following the pattern-based approach described in Section 4.2, the behaviour of the IB photo producer and the behaviour of the P2P Photo Sharing version 1 are decomposed into traces representing elementary behaviours. Then, the traces are analyzed and their basic mismatches are identified thanks to the basic mediators patterns. Subsequently, a composition strategy is applied to build elementary mediators, i.e., mediator traces, exploiting the basic mediators patterns. Finally, in order to automatically synthesize the behaviour of the whole Mediating Connector for the Photo Sharing applications, a composition approach aggregates the elementary mediators so to have a mediated coordination and communication.

Figure 4.10 shows the behaviour of the Mediating Connector for the applications considered in our example. We recall (as already sketched in Section 4.1) that the high level functionalities of the various applications are the following. Taking the producer perspective (1) *authentication* –for the IB producer only–, (2) *upload of photo*, and (3) *download*

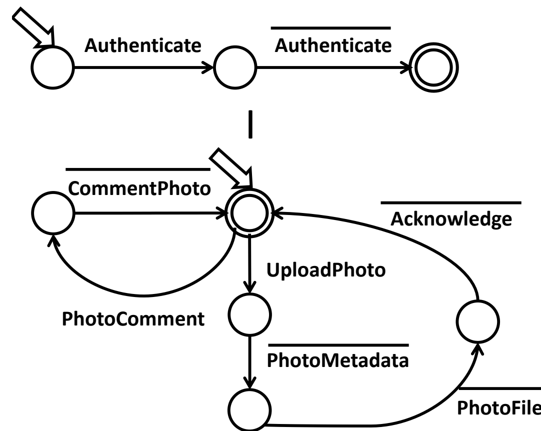


Figure 4.10: Behavioural description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 4.2 a) and P2P Photo Sharing version 1 of Figure 4.1 v1))

of comments, while taking the consumer perspective: (i) *download of photo*, and (ii) the *upload of comments*.

The mediator, in this example, allows the interaction between the two different Photo Sharing applications by (A) manipulating/translating and forwarding the conversations from one protocol to the other and (B) forwarding the interactions between the producer and its server. To better explain, in the following, we describe which Basic Mediator Patterns are used to detect and solve mismatches.

- The IB producer implements the authentication with the action “$\overline{Authenticate}$” while the P2P version 1 does not include such functionality, i.e., there is no semantically correspondent action in the P2P application (the complementary action is in the IB server – third parties communication). Then, in this case, the mediator has to forward the interactions from the producer to its server (case B above).

- The IB producer implements the upload of photo with the sequence of actions “$\overline{UploadPhoto}$. $Acknowledge$” where the former action sends both photo metadata and file and the latter models the reception of an acknowledgment. The corresponding download of photo implemented by the P2P version 1 is the sequence of actions “$PhotoMetadata$. $PhotoFile$”. Hence, although the actions are semantically equivalent, they do not synchronize. In order to detect/solve the mismatches, one has to use the basic patterns: *message splitting pattern* for the mismatch one send-many receive/many receive-one send “$\overline{UploadPhoto}$” vs. “$PhotoMetadata$. $PhotoFile$”; *message producer pattern* for the mismatch missing send/extra receive “$Acknowledge$” vs. no action. In this case, the mediator then (case A above) translates and forwards the conversations from one protocol to the other.

- The P2P version 1 implements the upload of comments with the action "$\overline{PhotoComment}$" while the IB producer implements the respective download of comments with the action "$CommentPhoto$". In order to detect/solve the signature mismatch "$\overline{PhotoComment}$" vs. "$CommentPhoto$", the *message translator pattern* is needed. Also, in this case (A above), the mediator translates and forwards the conversations from one protocol to the other.

4.6 CONCLUSION

In this chapter, we described the *Mediating Connector Architectural Pattern*, which by encapsulating the necessary support, is the key enabler for communication between mismatching components. Indeed, it solves the interoperability problems between heterogeneous and functionally compatible components.
Further, we described a set of *Basic Mediator Patterns*, including basic mismatches and their respective solutions, which specify the interoperability problems solved by the Mediating Connector Architectural Pattern.
Moreover we sketched a pattern-based approach to solve interoperability mismatches which allows: the component's behavior decomposition, the reasoning on mismatches, and the synthesis of a mediating connector behavior.

The patterns described above are twofold. On the one hand, the patterns are a set of design building blocks to tackle in a systematic way the protocol mediation problem. On the other hand, they rigorously characterize the kind of interoperability mismatches we deal with. Hence, one contribution of this chapter is to set the foundational base for the subsequent Chapters 5 and 6 that respectively present a theory of mediators and its extension.

As future works, we intend to: refine the design of the compositional approach based on patterns, that we sketched in Section 4.2, by exploiting the algebra of connectors presented in [6].
Moreover, in the direction of automated code generation, we also aim at providing: (i) the "concrete" Basic Mediator Patterns, i.e., the skeleton code corresponding to the "abstract" patterns presented in this chapter, (ii) the implementation of the pattern-based approach, i.e., the actual code for the component's behaviour decomposition and composition and the mediating connector behaviour building.

CHAPTER 5

MediatorS: A THEORY OF MEDIATING CONNECTORS

In our work, we want to approach the protocol interoperability problem in an automated way. Then, the problem we address here, is to automatically synthesize *mediators* that allow protocols to interoperate by solving their behavioral mismatches. To achieve such automation we designed a model-based synthesis technique that automatically derive a mediator by reasoning on protocols models, i.e., on their behavioural specification together with their semantic characterization.
As already mentioned, the key term *protocol*, defined in Section 1, will be used here to indicate *application-layer* protocols. Thus, in this Chapter we will address the *application-layer interoperability*.
While in Chapter 4 we have described the kind of mismatches we deal with in terms of the basic mediator patterns that can be used to solve them, in the following we describe the process that is performed in order to automatically elicit the mediator protocol. For the sake of exemplification we refer, also in this chapter, to the Photo Sharing scenario introduced in the previous chapter (Chapter 4). Still, among all the implementations that are sketched, we consider as reference scenario, the one made by: the IB Photo-Sharing Producer (Figure 4.2 a)) and the P2P Photo-Sharing version 1 (Figure 4.1 v1)).

This chapter specifically presents `MediatorS`, *a theory of mediating connectors* which is a revised and extended version of the theory presented in [62]. The acronym `MediatorS` summarizes the words `Mediator S`ynthesis.
This chapter is organized as follows. Section 5.1 introduces a formalization for interaction protocols, which paves the way for their abstraction, the automated reasoning about protocols functional matching and for the automated synthesis of mediators. Section 5.2 provides the algorithms implementing the three phases of our `AMAzING` process formalized in the previous section. Section 5.3 discuss the correctness of the mediator and finally Section 5.4 draws some conclusions.

5.1 FORMALIZING THE THEORY

In this section we describe the `MediatorS` theory, which is a formalization based on the `AMAzING` process. Figure 5.1 depicts the main elements of our methodology:

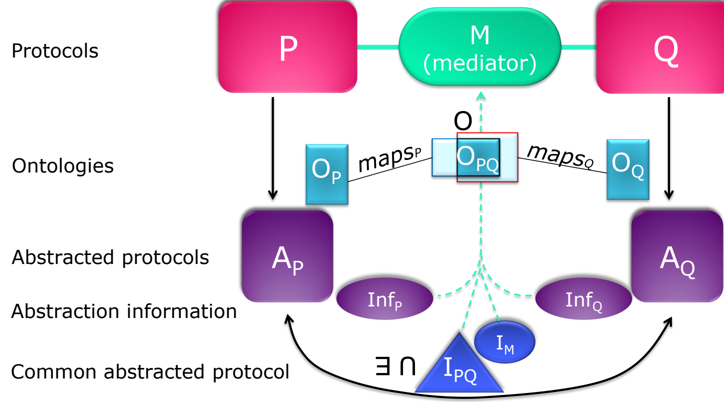


Figure 5.1: An overview of our approach

(i) Two application-layer protocols P and Q whose representation is given in terms of *Labelled Transition Systems* (*LTS*s), where the *initial* and *final states* on the LTSs define the *sequences of actions* (traces) that characterize the *coordination policies* of the protocols.

(ii) Two *ontologies* O_P and O_Q describing the meaning of P and Q's actions, respectively.

(iii) Two *ontology mapping functions* $maps_P$ and $maps_Q$ defined from O_P and from O_Q to a common ontology O. The intersection O_{PQ} on the common ontology identifies the "common language" between P and Q. For simplicity, and without loss of generality, we consider protocols P and Q that have disjoint languages and that are minimal where we recall that every finite LTS has a unique minimal representative LTS.

(iv) Then, starting from P and Q, and based on the ontology mapping, we build two abstractions A_P and A_Q by relabelling P and Q, respectively, where the actions not belonging to the common language O_{PQ} are hidden by means of silent actions (τ); moreover, we store some abstraction information (i.e., used to make the abstraction), Inf_P and Inf_Q, that in case of positive matching check, will be exploited to synthesize the mediator during the mapping;

(v) Then, we check the compatibility of the protocols by looking for complementary traces (the set I_{PQ} in figure), modulo mismatches and third parties communications, between the sets of traces T_P and T_Q generated by A_P and A_Q, respectively. If this is the case, then we are able to synthesize a mediator that makes it possible for the protocols to coordinate. Hence, we store the matching information (i.e., used to make the abstraction) I_M that will be exploited during the mapping.

(vi) Finally, given two protocols P and Q, and an environment E (not in figure), the mediator M that we synthesize is such that when building the parallel composition

$P|Q|E|M$, P and Q are able to coordinate by reaching their final states under the hypothesis of fairness.

Referring to the AMAzING process of Chapter 3, the proposed theory of mediators tackles all its steps. For our purposes, in this thesis we assume that: for each application, each device (e.g., PDA, smartphone, or tablet) is equipped with the behavioural specification of the application and a semantical characterization of its actions through ontologies. Taking the perspective of two systems that have compatible protocols and that also communicate with third parties, we also assume that there exists the proper environment for them, i.e., other systems representing third parties. Further, we concentrate on interoperability at application- and middleware-layer (respectively in this Chapter and the following) while assuming solved the heterogeneity of the underlying layers.

5.1.1 Abstraction Formalization

As already mentioned in Chapter 3, protocols abstraction is twofold: it makes protocols comparable thus allowing to perform the functional matching check and possibly reduces their size so to ease the automatic reasoning on them. If the protocols we deal with, let us say P and Q, are compatible being potentially able to communicate, we expect to find at a given level of abstraction a common (protocol) model C for them representing their potential interactions. This leads us to formally analyze such alike protocols to find such C –if it exists– and a suitable mediator that allows the interoperability that otherwise would not be possible. This problem can be formulated as a kind of antiunification problem [61, 96, 120, 98].
Let us consider: two extended LTSs P and Q modelling two protocols run by networked systems, two ontologies O_P and O_Q describing their actions, abstraction ontology mapping functions $maps_P$ on P and $maps_Q$ on Q (see Definition 5 in Section 1.2.2), and the intersecting common ontology O_{PQ} (i.e., the intersection among the mapped ontologies O_P and O_Q).

Infrastructure-based Photo-Sharing Producer	Common Language Projected on the Protocols (Complement Language)		Peer-to-peer Photo-Sharing version 1
$\overline{\text{UploadPhoto.}}$ Acknowledge	$\overline{UP}$ *(upload photo)*	*UP* *(download photo)*	PhotoMetadata. PhotoFile
CommentPhoto	*UC* *(download comment)*	$\overline{UC}$ *(upload comment)*	$\overline{\text{PhotoComment}}$
-	-	$\overline{UP}$ *(upload photo)*	$\overline{\text{PhotoMetadata.}}$ $\overline{\text{PhotoFile}}$
-	-	*UC* *(download comment)*	PhotoComment

Figure 5.2: Ontology mapping between Infrastructure-based Photo-Sharing Producer and the peer-to-peer Photo-Sharing version 1 (Figure 4.2 a and Figure4.1 v1 respectively)

In order to find the protocols' abstractions, we exploit the information contained in the ontology mapping to suitably relabel the protocols. Specifically, as detailed in the following, the relabelling of LTSs produces new LTSs that are labelled only by common actions and τ, and hence are more abstract than before (e.g., sequences of actions may have been compressed into single actions).
For illustration, Figure 5.2 summarizes the ontological information of the IB Producer of Figure 4.2 a) (first column) and of the P2P Photo-Sharing version 1 of Figure 4.1 v1) (third column). The second column shows their *common language*. We recall that: (1) the overlined actions are output/send action while non-overlined are input/receive; (2) the P2P application implements both roles, producer and consumer, while the IB application we are focusing on, is only the producer role (the overall Photo Sharing is implemented by three separate IB applications). This explains why we have in the table two non-paired actions; because they are paired with the actions of the other IB applications.
The *abstract protocols* are then obtained, leveraging on the abstraction ontology mapping (Definition 5 within Section 1.2.2), by relabelling protocols with labels of their common language and τ for the thirds parties languages. A necessary condition for the existence of a common language between two protocols P and Q, is that there exist two abstraction ontology mapping $maps_P$ on P and $maps_Q$ on Q that map the languages L_P^* of P and L_Q^* of Q into the same abstract/common ontology. Thus, to identify the common language, we first map each protocol's ontology into a common ontology and then by intersection, we find their common language.

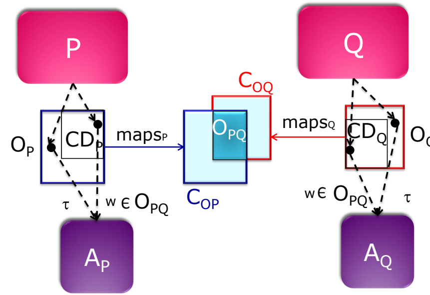


Figure 5.3: The abstract protocol building

Operationally, we do not work on protocols while we reason on *traces*: starting from a protocol P (Q resp.), we extract all the traces from it and apply the relabelling on the traces that result into a set of abstracted traces, with labels belonging to the common language and τ. However, the abstract protocol(s) of P (Q resp.), can be easily obtained by merging the traces where possible (e.g. common prefixes). Similarly, we use a reasoning on traces also for the matching and mapping phases.
It has to be noticed that the set of all the traces may not be finite. We consider minimal protocols[1]. Hence, the infinite set of traces is represented by a minimal automaton (containing at least a final state). Then, the abstraction ontology mapping on such minimal automaton, either applies directly (to the minimal automaton) returning a set of (abstracted)

[1]This is similar to the normal form of a system of recursive equations in [63] which is based on the idea to eliminate repetitions of equivalent recursive equations (that is equations with the same unfolding)

traces on the common language and τ, or it does not exist any automaton unfolding on which the abstraction ontology mapping applies.
Figure 5.3 depicts the abstraction of the protocols. We consider two minimal and deterministic protocols P and Q with their respective ontologies $O_P = (L_P^*, A_P)$ and $O_Q = (L_Q^*, A_Q)$ and their abstraction ontology mappings $maps_P$ and $maps_Q$ respectively. We first map O_P and O_Q, through $maps_P : L_P^* \to L$ and $maps_Q : L_Q^* \to L$ respectively, into a common ontology $O = (L, A)$ where C_{OP} and C_{OQ} represent the codomain sets of $maps_P$ and $maps_Q$ respectively.

The **common language** between P and Q is defined as the intersection O_{PQ} of C_{OP} and C_{OQ}. In particular, it is built by: (1) applying the abstraction ontology mapping to P and Q respectively thus obtaining the two sets of labels C_{OP} and C_{OQ} respectively; (2) starting from pairs of actions l and $\bar{l}$ ($\bar{l}, l$ resp.) belonging to C_{OP} and C_{OQ} respectively, store into O_{PQ} the action l –without taking into account the type output/input. Below, we define the common language more formally.

Definition 6 (*Common Language*) *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$*,*
- st_P, st_Q *be subtraces of traces of* P *and of* Q *respectively,*
- $O_P = (L_P^*, A_P)$ *be the ontology of* P *and*
  $O_Q = (L_Q^*, A_Q)$ *be the ontology of* Q*,*
- $O = (L, A)$ *be an ontology,*
- $maps_P : L_P^* \to L$ *be the abstraction ontology mapping of* P *and*
  $maps_Q : L_Q^* \to L$ *be the abstraction ontology mapping of* Q*.*

The common language O_{PQ} *between* P *and* Q *is defined as:*
$O_{PQ} = \{l : l \text{ (or } \bar{l}) = maps_P(st_P) \wedge l \text{ (or } \bar{l}) = maps_Q(st_Q)\ \}$
where st_P, st_Q *implement basic mismatches or combinations of them (as defined in Section 4.4 –see also Figure 4.4).* For instance, the pairs of labels ($\overline{UP}$, UP), or (UP, UP), or (UP, $\overline{UP}$), or ($\overline{UP}$, $\overline{UP}$) let us derive UP as an action belonging to the common language.

The **abstract protocol** A_P (A_Q resp.) of P (Q resp.), is built as follows:
for each trace t_P of P (t_Q of Q resp.) build a new trace t'_P (t'_Q) such that:

1. for each chunk (sequences of states and transitions) of t_P (t_Q resp.) labelled by subtraces on D_P (D_Q resp.), build a single transition in t'_P (t'_Q) labelled with a label on O_{PQ};
2. for all the other chunks of t_P (t_Q resp.) labelled with actions belonging to the third parties language, build chunks labelled with τ.

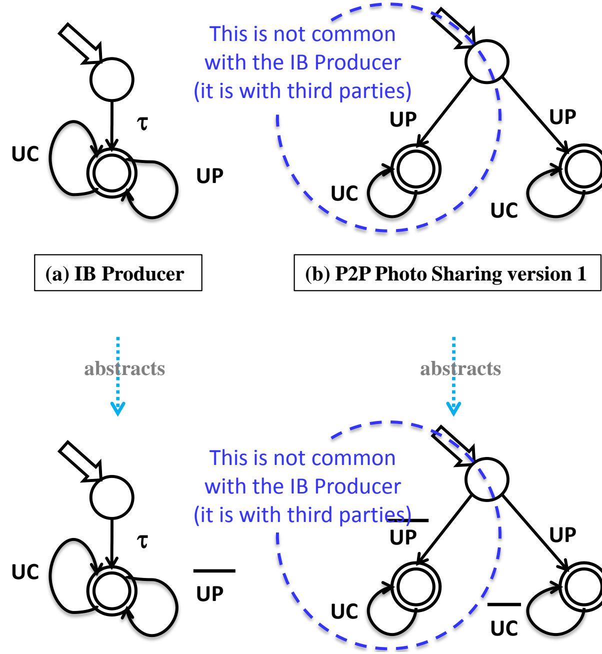


Figure 5.4: Abstracted LTSs of the Photo Sharing protocols

In the following we define more formally the relabelling function that we exploit:

Definition 7 (*Relabelling function*) *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$ *be protocols,*
- $O_P = (L^*_P, AX_P)$ *and* $O_Q = (L^*_Q, AX_Q)$ *be ontologies of* P *and* Q *respectively,*
- $O = (L, A)$ *be a common ontology for* P *and* Q*,*
- $maps_P : L^*_P \to L$ *and* $maps_Q : L^*_Q \to L$ *be abstraction ontology mappings of* P *and* Q *respectively,*
- C_{OP} *and* C_{OQ} *be the codomain sets of* $maps_P$ *and* $maps_Q$ *respectively,*
- O_{PQ} *be the common language between* P *and* Q*.*

The relabelling function $relabels$ *is defined as:* $relabels : (P, maps_P, O_{PQ}) \to A_P$
where $A_P = (S_A, L_A, D_A, F_A, s_{0_A})$ *and where*
$S_A \subseteq S_P$,
$L_A = \{l \in O_{PQ}\} \bigcup \{\tau\}$,
$D_A = \{s_i \xrightarrow{l} s_j \text{ (or } s_i \xrightarrow{\bar{l}} s_j) : \exists\, s_k \stackrel{w}{\Rightarrow} s_n \in D_P \wedge l \text{ (or } \bar{l}) = maps_P(w)\}$,
$F_A \subseteq F_P$, *and*
$s_{0_A} = s_{0_P}$.

The above definition applies similarly to Q: $relabels : (Q, maps_Q, O_{PQ}) \rightarrow A_Q$.
In the Photo Sharing scenario, the only label that is not abstracted in the common language is *authenticate* that represents a third party coordination. The IB producer and P2P Photo-Sharing version 1's abstracted LTSs are shown in Figure 5.4 where the upper part illustrates the protocols on the common language (i.e., common labels without taking into account output and input) while the bottom part of the figure illustrates the protocols on the common language projected on the protocols (i.e., labels where output and inputs are not abstracted). The subsequent step is to check whether the two abstracted protocols share a *complementary coordination policy*, i.e., whether the abstracted protocols may in fact synchronize, which we check over protocol traces as mentioned before.

5.1.2 MATCHING FORMALIZATION

The formalization described so far is needed to: (1) characterize the protocols and (2) abstract them into protocols on the same alphabet. Then we want to identify whether such two protocols are *functionally matching* and, if so, to synthesize the mediator that enables them to interoperate, despite behavioral mismatches and third parties communications. We recall that with *functional matching* we mean that given two systems with respective interaction protocols P and Q, ontologies O_P and O_Q describing their actions, abstraction ontology mapping functions $maps_P$ on P and $maps_Q$ on Q, and their intersecting common ontology O_{PQ}, there exists *at least one pair of complementary traces* (one trace in P and one in Q) that allows P and Q to coordinate towards a (common) goal. In other words, one or more sequences of actions of one protocol can synchronize with one or more sequences of actions in the other. This can happen by properly solving mismatches, using the basic patterns discussed in the previous chapter, and managing communications with third parties.
Then, to establish whether two protocols P and Q can interoperate, given their respective abstractions A_P and A_Q based on their common ontology O_{PQ} (i.e., common language) and possibly τ, we need to check that the abstracted protocols A_P and A_Q share complementary coordination policies. To establish this, we use the *functional matching relation* between A_P and A_Q, which succeeds if A_P and A_Q have *a set of pairs of complementary coordination traces*, i.e., at least one pair.
Before going into the definition of the compatibility or functional matching relation, let us provide the one of **complementary coordination policies**. Informally, two coordination policies are complementary if and only if they consist of the same set of complementary actions. We recall that *actions* are *complementary* iff they are the same action while having opposite output/input type. That is, traces t and t' are complementary if and only if they are the same set of actions while having opposite input/output type for all actions. More precisely: each output action (resp. input) of t has its complementary input action (resp. output) in t' and similarly with switched roles among t' and t. More formally:

Definition 8 (*Complementary Coordination Policies* or *Traces*) *Let:*

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$,

- *A_P, A_Q be the abstracted protocols of P and Q respectively,*
- *T_P and T_Q be the set of all the traces of A_P and A_Q, respectively,*
- *$t = l_1 l_2 \ldots l_n \in T_P$ and $t' = l'_1 l'_2 \ldots l'_m \in T_Q$.*

Coordination policies t and t' are complementary coordination policies iff the following conditions hold: discarding the τ,

(i) for each $l_i \in t \; \exists \; l'_j \in t' : l_i$ and l'_j are complementary actions (i.e., respectively output and input actions or input and output actions);

(ii) for each $l'_j \in t' \; \exists \; l_i \in t : l'_j$ and l_i are complementary actions (i.e., respectively input and output actions or output and input actions);

Note that (i) and (ii) above do not take into account the order in which the complementary labels l_i and l'_j are within the traces. Hence, two traces having all complementary labels (skipping the τ) but in different order are considered to be complementary coordination policies (modulo a reordering). Therefore, while doing this check, we store such information that will be used during the mediator synthesis in addition to other information, e.g., the abstraction information.
As said above, we perform the complementary coordination policies check on the abstracted protocols A_P and A_Q, which are expressed in a common language plus τ representing third parties synchronization. We use the **functional matching relation** to describe the conditions that have to hold in order for two protocols to be compatible. Formally:

Definition 9 (*Compatibility or Functional matching*) *Let:*

- *P and Q protocols,*
- *$relabels$ be a relabelling function,*
- *A_P and A_Q be the abstracted protocols, through $relabels$, of P and Q respectively, and*
- *t_i be a coordination policy of A_P and let t'_i be a coordination policy of A_Q.*

Protocols P and Q have a functional matching (or are compatible) iff there exists a set C of pairs $(t_i,\, t'_i)$ of coordination policies that results in complementary coordination policies.

Note that when considering applications that play only the client role, asking for services to a server, the functional matching definition above is slightly modified as follows: instead of checking the existence of a set of pairs of complementary traces, it checks the existence of a set of pair of traces that result in the *same* trace.

The *functional matching relation* defines necessary conditions that must hold in order for a set of networked systems to interoperate through a mediator. In our case, till now, the set is made by two networked systems and the matching condition is that they have at least a complementary trace modulo the τ. Such third parties communications (τ) can be just skipped while doing the check, but have to be re-injected while building the mediator. They hence represent information to be stored for the subsequent synthesis.

Generally speaking, protocols can also have more than one complementary trace, i.e., a set. We then define three different **levels of functional matching**, spanning from partial to total:

- **Intersection**: concerns cases where two protocols have only a subset of their traces that result in complementary coordination policies (from one trace to many, but not all);
- **Inclusion**: refers to the case in which two protocols have a shared set of complementary coordination policies and for one protocol this set coincides with the set of all its traces while for the other it represents a subset of all its traces;
- **Total Matching**: refers to the case in which two protocols have a shared set of complementary coordination policies and for both of them this set coincides with the set of all their traces.

5.1.3 Mapping Formalization

Consider two protocols P and Q that functionally match where the set C is made by their pairs of complementary coordination policies, and a protocol E representing the environment[2]. We want to synthesize a mediator M such that the parallel composition $P|M|Q|E$, allows P and Q to evolve to their final states. An action of P or Q can belong either to their *common language* or the *third parties language*, i.e., the language of the environment. Note that the environment is called third parties with respect to the interaction among P and Q. That is, taking the perspective of P, a protocol E is considered third party if P needs to interact with both E and Q in order to allow the communication among P and Q. This applies similarly to Q.
We build the mediator in such a way that it lets P and Q evolve independently for the portion of the behavior to be exchanged with the environment (denoted by τ action in the abstracted protocols) until they reach a "synchronization state" from which they can

[2]For the sake of simplicity, and without loss of generality, we consider only one protocol to be the environment but this can be generalized to an arbitrary number of protocols

synchronize on complementary actions. We recall that the synchronization cannot be direct since the mediator needs to perform a suitable translation according to the ontology mapping, e.g., $UC = CommentPhoto$ in one protocol and $\overline{UC} = \overline{PhotoComment}$ in the other.
As we said previously, operationally we work on traces instead of working on protocols. Hence we produce a set of mediating traces for the set C. We recall that C is made up of pairs of complementary coordination policies of the abstract protocols A_P and A_Q of P and Q respectively. Then, the mediator protocol AM for C can be easily obtained by merging the mediating traces. AM can be considered an "abstract mediator" since it mediates between abstract protocols. To obtain the corresponding "concrete mediator", we then need to translate each abstract action to its corresponding concrete (sequence of) action(s), i.e., on the languages of P and of Q.
Therefore, a **mediator** is a protocol that, for each pair $c_{ij} = (c_i, c_j)$ in C, builds a mediating trace m_{ij} such that, for each action (also τ) in c_i and in c_j it always first receives the action and then resends it. More formally:

Definition 10 (*Mediator*) *Let:*

- *C be the set of pairs of complementary coordination policies between two abstract protocols A_P and A_Q of protocols P and Q respectively;*
- *O_C be the common language among P and Q;*
- *$(c_i, c_j) \in C$ be a pair of complementary traces where $|c_i| = x$ $|c_j| = y$;*

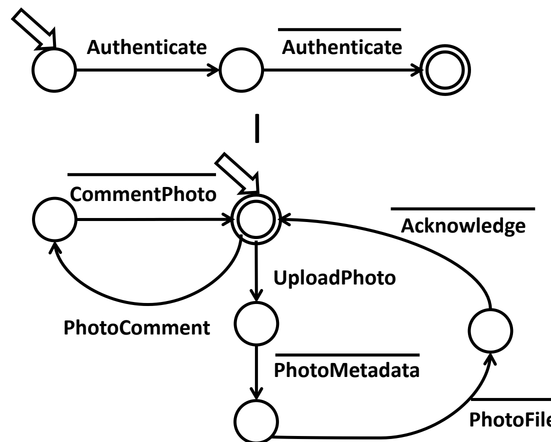
The mediator M for C is defined as follows:

$\forall\ (c_i, c_j)\ \exists$ *a mediating trace* $m_{ij} \in M : m_{ij} = l_1, l_2, \ldots, l_k \wedge k = x + y \wedge$
if $l_i = \overline{a}\ \wedge\ a \in O_C\ \wedge\ a \in c_i$ *then* $\exists\, 1 \leq h < x : l_h = a\ \wedge\ a \in c_j$;
if $l_i = \overline{a}\ \wedge\ a \in O_C\ \wedge\ a \in c_j$ *then* $\exists\, 1 \leq h < n : l_h = a\ \wedge\ a \in c_i$;

The mediator is logically made up of two separate components: M_C and M_T. M_C speaks only the common language and M_T speaks only the third parties language. M_C is a LTS built starting from the common language between P and Q whose aim is to solve the protocol-level mismatches occurring among their dual interactions (complementary sequences of actions) by translating and coordinating between them. M_T, if it exists, is built starting from the third parties language of P and Q and represents the environment. The aim of M_T is to let the protocols evolve, from the initial state or from a state where a previous synchronization is ended, to the states where they can synchronize again.

For illustration, we assume to have with the behavioral specification of the considered Photo Sharing applications, their coordination policies (thanks to the initial and final states on LTSs), their respective ontologies describing their actions, and the ontology mapping that defines the common language between IB producer and P2P Photo-Sharing version 1.

The first step is to *abstract* the protocols exploiting the ontology mapping. Following the theory, the abstracted protocols for the Photo Sharing scenario are illustrated in Figure 5.4. The second step is to check whether they have some complementary coordination policies. In this scenario, the IB producer is able to (complementary) simulate the P2P consumer, i.e., right branch of the LTS in Figure 5.4. The other branch, within the dashed circle has to be discarded since it is not common with the producer application (while being common with the server of the IB application). Then, the coordination policies that IB producer and P2P consumer share are exactly the consumer's ones. Hence, with the application of the theory to the scenario, we obtain the connector of Figure 4.10 presented in Chapter 4 that we report in the following.



Copy of Figure 4.10: Mediating Connector for the Photo Sharing

In this case, only the producer has third parties language actions and then the mediator is made by the part that translates and coordinates the common language and the part that simulates the environment by forwarding from and to it.

Note that the building of a connector can be slightly different according to the kind of protocols to be mediated.
If the control of a protocol P (and Q) is characterized by both output and input actions, then the mediator will (i) receive an action(s) from P, (ii) properly translate it, and (iii) send it to the Q and viceversa with exchanged roles between P and Q. The mediator will repeat these three steps several times to mediate each trace. Hence the mediator will synchronize with P (Q resp.) to receive or send messages.
Instead, if the control of protocol P (and Q) is only characterized by output actions, implementing the client role only, then the mediator will only receive actions from P (Q).

5.2 Implementing the Theory: Algorithms

In this section we describe algorithms to abstract the protocols (Section 5.2.1), check their compatibility (Section 5.2.2) and, if possible, synthesize a mediator (Section 5.2.3).

They provide an algorithmic description of the three steps process described in Chapter 3, and formalized in Section 5.1. The current version of the reasoning underlying the theory is not directly using the patterns while taking them into account and solving the problems they represent. Accordingly, the current version of the algorithms does not make explicit use of the patterns while detecting and solving their underlying problems. Pattern-based theory, and then pattern-based algorithms, are among future works. Technically, our algorithms work on traces instead of on LTSs despite the abstraction and the synthesis respectively takes as input and returns as output LTSs (corresponding to the traces sets).

5.2.1 Abstraction Algorithms

Considering two different protocols, we are looking for a common abstraction that makes them comparable and then enables the reasoning on them.

What. The *abstractionPhase* algorithm, described by Listing 5.1, takes as input two minimal and deterministic protocols P_1 and P_2, and their respective bijective abstraction ontology mappings $maps_1$ and $maps_2$ (see Definition 5). The algorithm automatically produces (line 14) the set of abstract traces T_1' and T_2' corresponding to the abstracted protocols of P_1 and of P_2, and a set of tuples TS summarizing the common language plus third parties language between P_1 and P_2. Each tuple is of the form $< l_1, m_1, m_2, l_2 >$ where l_1 and l_2 are subtraces of traces of P_1 and P_2 respectively and m_1 and m_2 are their corresponding labels on the common language plus tau.

```
1  Input: minimal and deterministic LTSs P1 and P2, bijective
     ↪abstractionOntologyMappingFunctions maps1 and maps2
2  Output: a triple <T1', T2', TS> with types TraceSet, TraceSet, TuplesSet respectively
3
4  <TraceSet,TraceSet,TuplesSet> abstractionPhase(LTS P1, LTS P2,
     ↪abstractionOntologyMappingFunction maps1, abstractionOntologyMappingFunction maps2
     ↪){
5      TraceSet T1,T2;
6          TuplesSet TS;
7          TraceSet T1',T2';
8          T1 := extractTraces(P1);
9          T2 := extractTraces(P2);
10         // we recall that P2 = (S2,L2,D2,F2,s02)
11         TS := buildCommonLanguagePlusTau(T1, T2, L2, maps1, maps2);
12         T1' := relabelTraces(T1, TS);
13         T2' := relabelTraces(T2, TS);
14         return <T1',T2', TS >;
15 }
```

Listing 5.1: abstractionPhase Algorithm

How. The abstraction phase algorithm is made up by three sub-phases:

1) traces extraction from protocols (lines 8-9 of Listing 5.1), which exploits the algorithm described by Listing 5.2 to extract the set of all the traces of a given protocol;

2) languages alignment of protocols (line 11 of Listing 5.1), which exploits the algorithm described by Listing 5.3 to build the common language plus tau between two traces sets (representing protocols);

3) traces relabelling (lines 12-13 of Listing 5.1), which exploits the algorithm described by Listing 5.4 to substitute subtraces of a given protocol with the correspondent abstract actions and τ through a given set of tuples TS recording the correspondences.

The $extractTrace$ Algorithm, takes as input a minimal and deterministic LTS P and returns as output the set of all the coordination policies it extracts from P. According to Definition 2 in Section 1.2.1, a coordination policy is a sequence of actions in an LTS from its initial state to one of its final states.

```
1  Input: minimal and deterministic LTS P
2  Output: TraceSet T of P
3
4  TraceSet T extractTraces(LTS P){
5      TraceSet T := ∅;
6      // we assume bounds on loops execution
7      while (∃ t: t is a coordination policy of P ∧ t ∉ T){
8          T := T ⋃ t;
9      }
10     return T;
11 }
```

Listing 5.2: extractTraces Algorithm

The **buildCommonLanguagePlusTau algorithm** takes as input two trace sets T_1 and T_2 representing two protocols and returns as output (line 23 of Listing 5.3) a set TS of tuples recording the common language plus tau built among T_1 and T_2 through the abstraction ontology mapping functions $maps_1$ and $maps_2$.

Specifically, the algorithm builds a quadruple in TS that may be of two kind:
(i) $< l_i, m_i, m_j, l_j >$ for each pair of subtraces l_i and l_j of T_1 and T_2 respectively that, through $maps_1$ and $maps_2$, result to be semantically corresponding, with labels m_i and m_j on the common language plus tau (lines from 7 to 12 of Listing 5.3). Note that m_i and m_j are the same label with opposite type input/output and are called common (complementary) names or abstract actions.
(ii) $< l_i, \tau_h, -, - >$ (or $< -, -, \tau_k, l_j >$ respectively) for each remaining subtrace, i.e., the subtraces that do not have correspondences (lines from 13 to 17 and from 18 to 22 respectively of Listing 5.3). The τ represent conversations exchanged with third parties and hence abstract only the actions of one protocol (the τ are not common indeed we use subscripts to distinguish the τ of one protocol with respect to the ones of the other -see also lines 14 and 20).

```
1  Input: TraceSets T1 and T2 (of LTSs P1 and P2 respectively), labelSet L2,
       ↪abstractionOntologyMappingFunctions maps1 and maps2
2  Output: TuplesSet TS
3
4  pairsSet buildCommonLanguagePlusTau(TraceSet T1, TraceSet T2, labelSet L2,
       ↪abstractionOntologyMappingFunction maps1, abstractionOntologyMappingFunction maps2
       ↪) {
5      TuplesSet TS := empty;
6          quadruple qij, qi, qj := empty;
7      forEach trace ti ∈ T1{ // this covers all the traces of T1
```

```
8        forEach (l_i := subtrace(t_i)) : ∃l_j ∈ L_2^* ∧ ((m_h := maps_1(l_i) ∧ m̄_h := maps_2(l_j)) ∨
             ↪(m̄_h := maps_1(l_i) ∧ m_h := maps_2(l_j))) { // this covers all the subtraces of
             ↪traces in T_1 that belong to the common language
9            q_ij := < l_i, m_h, m̄_h, l_j > or < l_i, m̄_h, m_h, l_j > (accordingly)
10           // m_h ranges on m_1, m_2, ..., m_k;
11           TS := TS ⋃ q_ij;
12       }
13       forEach remaining l_i := subtrace(t_i){ // this covers all the subtraces of traces in
             ↪T_1 not already considered that belong to third parties conversation
14           q_i := < l_i, τ_h, −, − > // h ranges over 1, 2, ..., k;
15           TS := TS ⋃ q_i;
16       }
17   }
18   forEach (l_j ∈ L_2^* : l_j has not already been considered){// this covers all the
         ↪subtraces of traces in T_2, not already covered, that belongs to third parties
         ↪conversation
19           q_j := < −, −, τ_h, l_j >
20           // h ranges over k+1, k+2, ..., k+n;
21     TS := TS ⋃ q_j;
22     }
23 return TS;
24 }
```

Listing 5.3: buildCommonLanguagePlusTau Algorithm

The **relabelTraces algorithm** relabels/rewrites/abstracts the traces of T by collapsing, where possible, sequences of states and transitions (see also the abstract protocol building in Section 5.1.1 before Definition 7). It takes as input a trace set T and a tuple set TS recording correspondences/mappings of subtraces of T (with respect to another protocol T_O) and returns a relabelled traces set T_1' of T (line 14 of Listing 5.4).

For each trace $t_i \in T$ (line 6), the algorithm substitutes subtraces st_j of t_i with their corresponding abstract actions belonging to TS (lines 8 to 10). The corresponding actions can belong either to the common language ranging among $m_1, m_2, \ldots, m_n$ or can belong to third parties languages and range among $\tau_1, \tau_2, \ldots, \tau_m$.

```
1 Input: TraceSet T, TuplesSet TS
2 Output: TraceSet T'
3
4 TraceSet relabelTraces(TraceSet T, TuplesSet TS) {
5     Trace t_ir;
6     forEach trace t_i ∈ T {
7         t_ir := t_i;
8         forEach st_j := subtrace(t_ir) : m̄_i (or m_i or τ_i) corresponds to st_j in TS i.e.,
              ↪< st_j, m̄_i, m_i, st_k >∈ TS (or < st_j, m_i, m̄_i, st_k >∈ TS or < st_j, τ_i, −, − >∈ TS){
9             t_ir := rewrite(t_ir, m̄_i)(or t_ir := rewrite(t_ir, m_i) or t_ir := rewrite(t_ir, τ_i) accordingly)
10        }
11        if t_ir ≠ t_i then
12            T' := T' ⋃ t_ir;
13    }
14 return T';
15 }
```

Listing 5.4: relabelTraces Algorithm

5.2.2 MATCHING ALGORITHMS

What. The *(semantic) matching phase* is described by Listing 5.5. It checks the existence of (at least) a complementary coordination policy between two traces sets T_1' and T_2' taken as input. The algorithm returns as output (line 40 of Listing 5.5): *matching*, indicating whether a matching exists or not, *relation* specifying the levels of (functional) matching, *matchingTraces* which is the set of complementary coordination policies.
matching can respectively be `yes` (line 25) if at least a matching trace is found or `no` otherwise (line 24). In case *matching* = `yes`, *relation* can be either `intersection` (lines 28 to 30) or `inclusion`, i.e., `t1containst2` (lines 31 to 33) or `t2containst1` (lines 34 to 36), or it can be `totalmatching` (lines 37 to 39). In case *matching* = `no` the relation is the string `*`.

```
1  Input: TraceSet T1' and TraceSet T2'
2  Output: String matching, String relation, triplesSet matchingTraces
3
4
5  < String, String, triplesSet > matchingPhase(TraceSet T1', TraceSet T2') {
6  Boolean intersect1 := false;
7  Boolean intersect2 := false;
8  Boolean intersection := false;
9  Boolean t1containst2 := false;
10 Boolean t2containst1 := false;
11 Boolean totalmatching := false;
12 String matching := '';
13 Boolean relation := '*';
14 triplesSet checkedTraces := emptyset; // contains triples <ti,tj,resp> : ti ∈ T1', tj ∈ T2',
     ↪resp ∈ {true,false}
15 triplesSet matchingTraces := ∅;
16 for each Trace ti ∈ T1' {
17     for each Trace tj in T2' {
18         // element is of the kind <ti,tj,detailsn> : ti ∈ T1', tj ∈ T2', details is a data
               ↪structure which, among other fields, includes resp ∈ {true,false}
19         triple element := checkCompatibility (ti, tj);
20         checkedTraces := checkedTraces ⋃ element;
21     }
22 }
23 matchingTraces := {<ti,tj,detailsn>∈ checkedTraces : detailsn.resp = true };
24 if (matchingTraces = ∅) then matching := NO
25 else matching := YES;
26 if (∃ti' ∈ T1' : ∄ elementk = ti' ∈ matchingTraces) then intersect1 := true;
27 if (∃tj' ∈ T2' : ∄ elementk = tj' ∈ matchingTraces) then intersect2 := true;
28 if (intersect1 = true and intersect2 = true) then
29     intersection := true;
30     relation := INTERSECTION;
31 else if (intersect1 = true) then
32     t1containst2 := true;
33     relation := T1CONTAINST2;
34 else if (intersect2 = true) then
35     t2containst1 := true;
36     relation := T2CONTAINST1;
37 else if (intersect1 = false and intersect2 = false) then
38     totalmatching := true;
39     relation := TOTALMATCHING;
40 return < matching, relation, matchingTraces >;
41 }
```

Listing 5.5: matchingPhase Algorithm

How. For each trace $t_i \in T'_1$ and each trace $t_j \in T'_2$ (lines 16 to 22), the matchingPhase algorithm exploits an auxiliary procedure, shown by Listing 5.6, to perform the compatibility check between two traces.

The $checkCompatibility$ Algorithm reasons on two traces t_1 and t_2 taken as input and finds, if any, their subtraces correspondences. The triple it returns, $< t_1, t_2, details >$, is made up by t_1 and t_2 and $details$ which is a data structure recording the correspondences found. Among other fields it includes $resp$, ranging on $\{true, false\}$, indicating whether or not the two traces are matching.

```
1 triple checkCompatibility(Trace t1, Trace t2)
```

Listing 5.6: checkCompatibility Algorithm

5.2.3 Mapping Algorithms

What. The *mapping phase*, described by Listing 5.7, builds the mediator behavior. It takes as input $matching$, $relation$, and $matchingTraces$ coming from the previous phase, and the TuplesSet (describing the common language plus the third parties language) coming from the abstraction phase. The algorithm returns as output $mediator$ (line 50 of Listing 5.7) which can be a non-empty LTS in case $matching$ is `yes` and an empty LTS in case $matching$ is `no` (line 12).

```
1 Input: String matching, String relation, triplesSet matchingTraces, pairsSet PS (output of
    ↪abstractionPhase,i.e., buildCommonLanguagePlusTau)
2 Output: LTS mediator
3
4 LTS mappingPhase (String matching, String relation, triplesSet matchingTraces, TuplesSet TS
    ↪) {
5 triple triple_aux := empty; // of the kind <t_i,t_j,details_n>
6 triple reorderedTraces := empty; // of the kind <t_i,t_j,details_n>
7 LTSSet medt := empty LTS set;
8 LTSSet media := empty LTS set;
9 LTS med_nontau :=empty LTS;
10 Trace concrete_1 := '';
11 boolean reordered := false;
12 if (matching = NO) then return emptyLTS;
13 else if (matching = YES) then {
14         // matchingTraces contains triples <t_i,t_j,details_n>
15         forEach triple_i ∈ matchingTraces {
16            triple_aux := triple_i;
17            if (checkToReorder(triple_aux) = true) then {
18               reorderedTraces.details.mediator := reorder(triple_aux);
19               reordered := true;
20            } else {
21               triple_aux.details.mediator := computeMediator(triple_aux);
22               reordered := false;
23            }
24             forEach action act = τ_h ∈ t_i ∈ triple_aux {
25                 concrete_1 := concretizeAction(τ_h, TS);
26                 medt := medt ⋃ buildLts(concrete_1);
27             }
28             forEach action act = τ_k ∈ t_j ∈ triple_aux {
29                 concrete_1 := concretizeAction(τ_k, TS);
```

```
30                  medt := medt ⋃ buildLts(concrete_1);
31              }
32              if (reordered = true) then {
33                  forEach action m_i ∈ reorderedTraces.details.mediator {
34                      concrete_1 := concretizeAction(m_i, TS);
35                      // m_i can either be an INPUT or an OUTPUT action and can either
                            ↪belong to a protocol or to the other hence resulting translated
                            ↪ into one language or the other accordingly
36                      med_nontau := append(buildLts(concrete_1), med_nontau);
37                  }
38              } else {
39                  forEach action m_i ∈ triple_aux.details.mediator {
40                      concrete_1 := concretizeAction(m_i, TS);
41                      med_nontau := append(buildLts(concrete_1), med_nontau);
42                  }
43              }
44          }
45          media := medt ⋃ med_nontau;
46  }
47  LTS mediator := parallelComposition(media);
48  mediator := makeDeterministic(mediator); // all its ltss
49  mediator := makeMinimal(mediator);
50  return mediator;
51  }
```

Listing 5.7: mappingPhase Algorithm

How. In the case $matching$ is `yes` (line 13), i.e., a matching exists, the algorithm performs the following operations. For each matching traces pair (line 15)
(i) if needed, it computes and stores into `reorderedTraces` a mediator which properly reorders the non-tau actions of the two traces (lines 18 and 19). Otherwise computes and stores a proper mediator without reordering (lines 20 to 23).
(ii) For each tau action of a trace (lines 24 to 27) and of the other trace (lines 28 to 31), concretizes them with their actual/corresponding actions in TS and builds a proper mediator LTS for them.
(iii) Concretizes and builds the proper LTS for the mediators in the point (i).
In the case a reordering has been performed (lines 32 to 37), the algorithm concretizes -according to the proper protocol language- the non-tau actions of the mediator created and stored in reorderedTraces and build the corresponding LTS.
In the case the reordering is not needed (lines 39 to 44), the algorithm concretizes the mediator stored in $triple_{aux}$ -according to the proper protocol language- and build the corresponding LTS.

5.3 Correctness Discussion

In the previous sections we presented a theory to synthesize mediators for protocol interoperability. Here we discuss the correctness-by-construction of the produced mediator, i.e., that *all the mediator traces allow a correct protocols interaction*. We will demonstrate the correctness by absurd.

As mentioned in the previous chapters, we work under some assumptions that we recall in the following.

Let us consider two compatible protocols P_1 and P_2, and a protocol P_3 all working under a fairness hypothesis. P_3 is able to directly interoperate with P_1, thus being a third party when taking the interoperability perspective of the pair P_1 and P_2. Let us also call A_1 and A_2 the abstracted protocols of P_1 and P_2 respectively.
We assume:

- A1 to know the correct behavioural description of the protocols as LTS, i.e., correctly representing the behaviour of the real/running protocol;
- A2 to know the correct (i.e. actual) semantical characterization of the protocol's actions, i.e., of the ontological description of the labels on the LTS;
- A3 the existence of the correct environment where protocols communicate also with it, i.e. proper third parties, other protocols;
- A4 to know the correct bijective abstraction ontology mapping functions of the protocols ontology;
- A5 to use a correct algorithm to translate a set of traces into the corresponding LTS that represent them;

Considering P_1, P_2, and P_3 under the above assumptions, the theory (1) identifies the set C of matching coordination traces between A_1 and A_2 and (2) creates a mediator M for P and Q.
To build M, as mentioned in Section 5.1.3, we first build an abstract mediator AM for C, mediating between abstract traces, and then we translates the abstract actions of AM into concrete actions of P_1, P_2 and P_3.
Thus, to show the correctness of the mediator M we can show the correctness of the mediator AM, i.e., that *each mediator trace is correct* allowing a correct protocols interaction. This backward step from M to all the traces of AM is possible because of assumptions A4 and A5 that respectively guarantee that: the inverse function of the abstraction ontology mapping applied to go from M to AM is correct, and the algorithm to translate the set of mediating traces into the corresponding AM representing them is correct.

A correct protocols interaction is defined as follows.
Let us consider two protocols P and Q and two traces t_P of P and t_Q of Q.
t_P and t_Q represent a **correct protocol interaction** between P and Q if they are the same sequence of actions with opposite output/input type hence being able to synchronize reaching one of their respective final states.

In our case the mediator has to mediate among matching traces. Then let us consider a pair of matching traces t_1 and t_2.
Given the above, a *mediating trace* m is **correct** for t_1 and t_2 if it allows a correct protocol interaction.
We recall that this means that t_1 and t_2 implement a complementary functionality while possibly having mismatches and interleaved third parties actions. Thus a correct protocols

interaction is a traces synchronization that allows t_1 and t_2 to evolve until they reach one of their respective final states by communicating, through the mediator, either together (on the complementary functionality) or with third parties. This means that m, in order to be correct, must allow such kind of synchronization.

But this is exactly how we build our set of mediator traces. In particular, for each pair of matching traces we build a mediator trace that let them evolve together by mediating their mismatches and forwarding the third parties interactions hence being able, under the described assumptions, to let two matching traces correctly communicate.
Summarizing, each mediator trace we build is correct-by-construction and hence the whole mediator is correct.

Demonstration by absurd. Let us suppose by absurd that there exists a non-correct mediating trace m_{ij} of the mediator for the pair of matching traces t_i and t_j. That is, m_{ij} does not allow to evolve until its final state at least one among t_i and t_j. This can be translated into three cases: (i) only t_i does not evolve to its final state, (ii) only t_j does not evolve to its final state, and (iii) both t_i and t_j do not evolve until their final states.

Case (i). Only t_i does not evolve to its final state.
This could happen because (1) an action of t_i has to be exchanged with third parties and it does not happen, or (2) an action of t_i belongs to the common language and has to be exchanged with t_j, but this does not happen. In the case (1), either the action is erroneously categorized as third parties action or m_{ij} is erroneous. But this violates the assumptions A2 and A3 that ensure that the action is an actual third parties action, and conflicts with the conditions in the definition to build m_{ij} that, starting from t_i and t_j under the assumption A1, ensures that the action is exchanged performing a receive of this action followed by the corresponding send.
In the case (2), either the action is erroneously categorized as common language action or m_{ij} is erroneous. The demonstration is straightforward from case (i) (1).

Case (ii). Only t_j does not evolve to its final state.
The demonstration is straightforward from case (i).

Case (iii). Both t_i and t_j do not evolve until their final states. This means that both t_i and t_j are blocked on an action that prevents them to evolve. We can have two cases: (a) the pair of actions of t_i and t_j is of the form (τ_i, τ_j) (b) the pair of actions is made by actions of the common language that should be exchanged with the other trace but this does not happen. The case (a) is demonstrated by absurd from the case (i)(1). The case (b) can depend on two things: either the actions are erroneously categorized as common language actions or the reordering performed by m_{ij} is erroneous. The former hypothesis is demonstrated by absurd following the demonstration of case (i)(2), while the latter conflicts with the conditions in the definition to build m_{ij} that, starting from t_i and t_j under the assumption A1, ensures that the action is exchanged performing a receive of this action followed by the corresponding send.

5.4 CONCLUSION

In this chapter, we described our proposed theory for the interoperability of application-layer protocols that are observable at the interface level. Key issue is to solve behavioral mismatches among the protocols although they are functionally matching.

The proposed theory is a means to: (1) clearly define the problem, (2) show the feasibility of the automated reasoning about protocols, i.e., to check their functional matching and to detect their behavioral mismatches, and (3) show the feasibility of the automated synthesis of abstract mediators under certain conditions to dynamically overcome behavioral mismatches of functionally matching protocols. We have also shown the correctness of the synthesized mediator.

Our theoretical framework is a first step towards the automated synthesis of actual mediators. As detailed in the next chapter, significant part of our current work is on leveraging practically the proposed theory in particular dealing with automated reasoning about protocol matching and further automated protocol mapping. We are also concerned with the integration with complementary work so as to develop an overall framework enabling the dynamic synthesis of emergent connectors among networked systems.

Relevant effort includes the study of: learning techniques to dynamically discover the protocols that are run in the environment, middleware protocols mediation, dependability assurance, data-level mediation, as well as algorithms and run-time techniques towards efficient synthesis.

First results of such integration effort are described in Chapter 6 that reports about the extension of the `MediatorS` theory to deal also with middleware mediation and data, and in paper [18] that illustrates the combination of the automated synthesis technique described in this chapter with a monitoring mechanism.

CHAPTER 6

EXTENDING THE MediatorS THEORY TO ENCOMPASS MIDDLEWARE-LAYER

The previous two chapters respectively presented a mediator synthesis process called AMAzING and a theory of mediators implementing such process called MediatorS. As seen, the mediator synthesis relies on:

1. the abstraction of LTSs protocols whose actions semantics is given using ontologies;
2. the definition of a functional matching relation over abstract protocols to identify whether they may coordinate to achieve a common goal using a mediator that solves their possible behavioral mismatches;
3. the definition of protocol mapping which returns a mediator implementing appropriate solutions to possible mismatches between the protocols; in other words, the mediators compose basic mediation patterns introduced in Chapter 4.

Since both the process and the theory that we proposed are general, in this chapter we will extend the interoperability scope by considering together *application-layer* and *middleware-layer interoperability*. Our aim is then to apply AMAzING and MediatorS on both application- and middleware-layers. Thus with the term *protocol* in this chapter we refer to a combination of application- and middleware-layer protocol plus the data conveyed. In this chapter we illustrate:

- a refined model of networked systems that: (i) makes explicit their high-level functionalities whose aim is to reduce the complexity of protocol matching verification; (ii) extends the protocol model so to include, together with the application actions, also the middleware functions and input/output data;
- the definition of (a) a middleware ontology (b) a middleware alignment and (c) a middleware abstraction;

The above listed extensions lead to refine the working of protocol abstraction, functional matching and mapping, defined in the previous chapter.

In the following, we present a revised version of our work [17] where: Section 6.1 introduces the refined model of networked systems (Section 6.1.1) including also the extended protocol model (Sections 6.1.2 and 6.1.3). Section 6.2 describes ontologies that are used to conceptualize middleware (Section 6.2.1) in addition to application and data (Section 6.2.2). Then, Sections 6.3, 6.4, and 6.5 respectively discuss the refinement of the abstraction, of the functional matching relation and of the mapping introduced in the previous chapter taking into account the above extensions. Finally, Section 6.6 concludes with a summary of the chapter's contributions and future work. Our contribution primarily lies in dealing with interoperability/mediation of both application and middleware layer.

6.1 Modeling Networked Systems

In this section, towards reducing the complexity of protocol matching verification and to model middleware functions and input/output data together with the application actions, we refine the model of the networked systems provided so far.

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 [131]. Still, a crucial issue is which description of resources should be advertised, which ranges from simple (attribute, value) pairs as with SLP[1] to advanced ontology-based interface specification [13].

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 this functionality. In other words, networked systems must advertise the *high-level functionalities* they provide and/or consume to be able to meet according to them. We call such functionalities **affordances** (Section 6.1.1). A necessary condition for the compatibility/composability of networked system is *affordance matching* (Section 6.4.1). That is a networked system requires an affordance that *matches* an affordance provided by another. In the theory of mediators introduced in the previous chapter, affordances are not explicit and are inferred from the reasoning on the set of complementary coordination policies of the networked systems' abstract protocols. To reduce the complexity of checking networked systems compatibility, we consider explicit the specification of the networked systems' affordances, in a way similar to the specification of capabilities in the definition of semantic Web services. In this way we can first perform an high-level check, i.e. affordance matching. If this is successful we proceed with the reasoning otherwise we stop.

Then, the *specification of networked systems decomposes into a number of affordances whose behaviors are defined as* **protocols**. We recall that we consider the observable protocol, i.e., sequences of actions visible at the interface level exchanged with other systems. Protocols actions are specified as part of the system's **interface signature** (Section 6.1.2) while the modeling of protocols relies on some concurrent language or equivalently *LTSs* and may be advertised by the system or be possibly learned (Section 6.1.3). Finally, the semantics of actions is defined by exploiting **ontologies**.

[1] http://www.openslp.org/

6.1.1 AFFORDANCE

An *affordance* denotes a high-level functionality provided to or required from the networked environment. We model an affordance as a tuple:

$$Aff = <\text{Type, Op, I, O}>$$

where:
Type denote that either an affordance is offered/provided in the network (*Prov*) or consumed/required (*Req*) or both required and provided (*Req_Prov*). This latter is common in peer-to-peer systems;
Op gives the semantics of the functionality associated to the affordance in terms of an ontology concept;
I (resp. *O*) specifies the set of inputs (resp. outputs) of the affordance, and is defined as a tuple $< i_1, \ldots, i_n >$ (resp. $< o_1, \ldots, o_m >$) with each i_h (resp. o_k) being an ontology concept.

As an illustration, the *consumer* affordance of the Photo Sharing scenario is defined as: *Photo-Sharing_Consumer* = *<Req, Photo-Sharing_Consumer, <PhotoComment>, <Photo>>* where the meaning of concepts is direct from their names (see further Section 6.2 for the definition of the ontology).

6.1.2 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 affordances 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 tied 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 tuples as defined in the following. More formally:

$$Interface = \{< m, a, I, O >\}$$

where:
m 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 6.2, the tuple elements are ontology concepts so that their semantics may be reasoned upon.
As an illustration, the listing[2] below 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[3] functions, i.e., the client-side application actions are invoked though SOAP middleware using the *SOAP-RPCInvoke* function, while they are processed on the server side using the two functions *SOAP-RPCReceive* and *SOAP-RPCReply*. The specific applications actions are rather straightforward from the informal sketch of the scenario in Sections 1.1 and 4.1. 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.

```
1  Interface_photo_sharing_producer = {
2    <SOAP-RPCInvoke, Authenticate, login, authenticationToken>
3    <SOAP-RPCInvoke, UploadPhoto, photo, acknowledgment>
4  }
5  Interface_photo_sharing_consumer = {
6  <SOAP-RPCInvoke, SearchPhotos, photoMetadata, photoMetadataList>
7  <SOAP-RPCInvoke, DownloadPhoto, photoID, photoFile>
8  <SOAP-RPCInvoke, DownloadComment, photoID, photoComment>
9  <SOAP-RPCInvoke, CommentPhoto, photoComment, acknowledgment>
10 }
11 Interface_photo_sharing_server = {
12 <SOAP-RPCReceive, Authenticate, login, ∅>
13 <SOAP-RPCReply, Authenticate, ∅, authenticationToken>
14 <SOAP-RPCReceive, UploadPhoto, photo, ∅>
15 <SOAP-RPCReply, UploadPhoto, ∅, acknowledgment>
16 <SOAP-RPCReceive, SearchPhotos, photoMetadata, ∅>
17 <SOAP-RPCReply, SearchPhotos, ∅, photoMetadataList>
18 <SOAP-RPCReceive, DownloadPhoto, photoID, ∅>
19 <SOAP-RPCReply, DownloadPhoto, ∅, photoFile>
20 <SOAP-RPCReceive, DownloadComment, photoID, ∅>
21 <SOAP-RPCReply, DownloadComment, ∅, photoComment>
22 <SOAP-RPCReceive, CommentPhoto, photoComment, ∅>
23 <SOAP-RPCReply, CommentPhoto, ∅, acknowledgment>
24 }
```

Listing 6.1: Interface signature

The peer-to-peer-based implementation defines a single interface signature, as all the peers feature the same observable actions. It further illustrates the naming of actions over domain data types of the application data instead of operations since the actions are

[2]As defined in the next section, *photoFile* and *photoComment* include photoID.
[3]http://www.w3.org/TR/soap/

data-centric and are performed through functions of the LIME[4] tuple-space middleware.

```
1  Interface_photo_sharing = {
2   <Out, PhotoMetadata, ∅, photoMetadata>
3   <Out, PhotoFile, ∅, photoFile>
4   <Rdg, PhotoMetadata, photoMetadata, photoMetadataList>
5   <Rd, PhotoFile, photoID, photoFile>
6   <Rd, PhotoComment, photoID, photoComment>
7   <Out, PhotoComment, ∅, photoComment>
8   <In, PhotoComment, photoID, photoComment>
9   <Rd, PhotoComment, photoID, photoComment>
10 }
```

Listing 6.2: Photo sharing

Listing 6.2 highlights four primitives: a non-blocking write operation (Out) to produce a tuple writing it into tuple space; a blocking non-destructive read (Rd) to get a copy of *one* tuple and one to get *all* the tuples (Rdg) matching a template from the tuple space; a blocking destructive read that getting a copy consumes/removes it from the tuple space (In).

6.1.3 Affordance Protocol

Given the networked system's interface signature, the behavior of the system's affordances is specified as protocols over the system's actions, which are defined in the interface signature. Such protocols may be explicitly defined using some concurrent language or the equivalent LTSs, as part of the networked system's advertisements, as for instance promoted by Web services languages. Alternatively, the protocol specification may be learned in a systematic way based on the system's interfaces but this is beyond our scope. Thus, in this chapter, we assume that protocols are explicitly advertised. Different languages may be considered for such a specification from formal modeling to programming languages. To provide a concrete example, we exploit today's well-established language from the Web service domain, i.e., BPEL[5]. 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 and the corresponding LTS that allows abstracting BPEL processes for automated reasoning [53]; and (iii) the rich tool set coming along with BPEL, which in particular eases process definitions by developers. However, in a way similar to the definition of interface signatures, the language must allow specifying communication actions using the various communication paradigms enabled by today's middleware solutions and not only those promoted by Web service technologies. Precisely, BPEL needs to be enriched so as to support interaction with networked systems using different interaction patterns and protocols, i.e., other than message-based ones that are classically associated with Web services, which can be addressed in a systematic way using the BPEL extension mechanism.

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

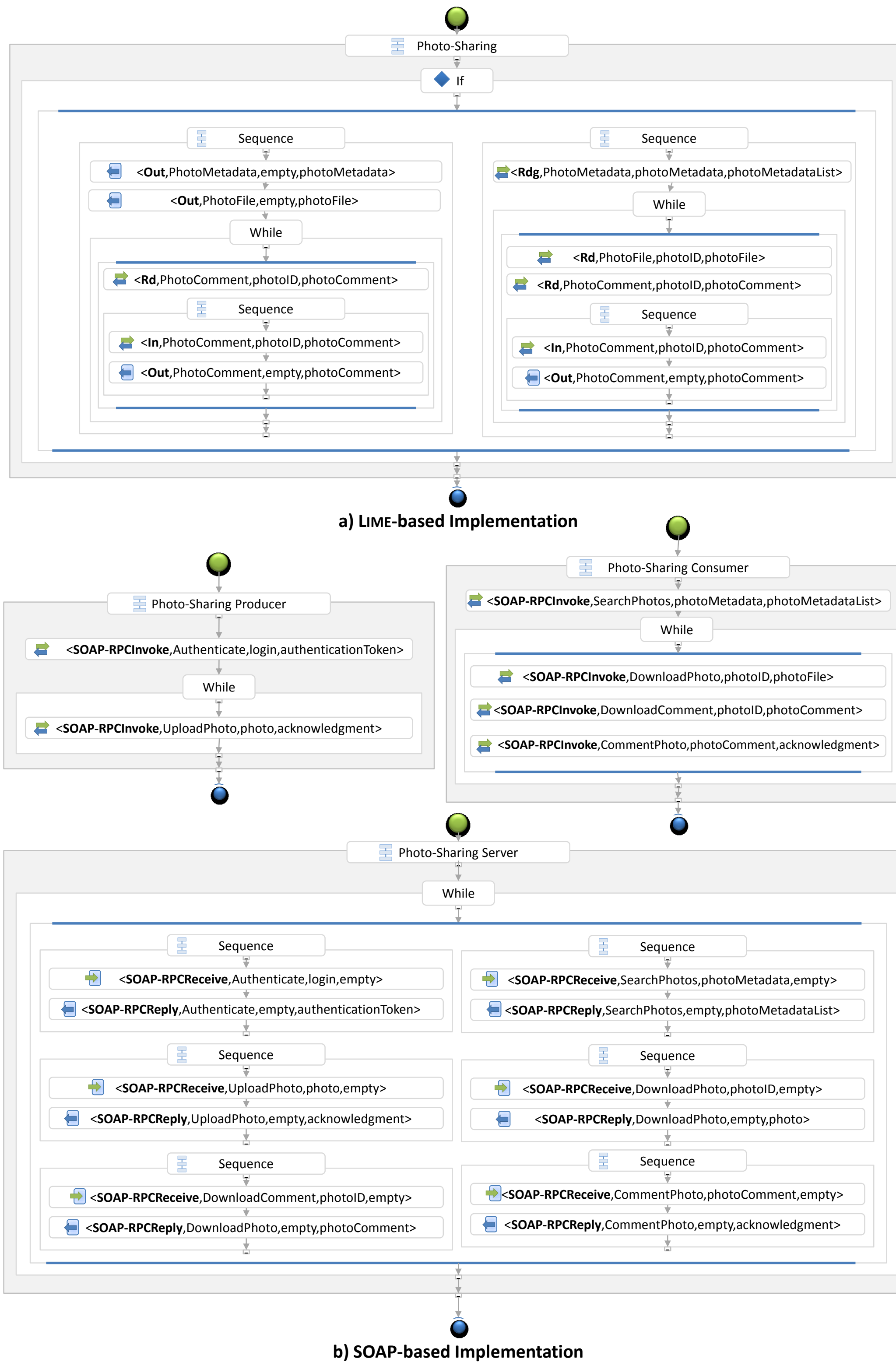


Figure 6.1: Infrastructure- and peer-to-peer-based photo sharing

For illustration, Figure 6.1 gives the specification of the protocols associated with the peer-to-peer and infrastructure-based Photo Sharing applications where we more specifically consider: (a) LIME-based peer-to-peer and (b) SOAP-based infrastructure-based implementations of the Photo Sharing application. The protocol executed by LIME-based networked systems 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, connectors shall enable seamless interaction of the LIME-based Photo Sharing implementation with systems implementing affordances of the infrastructure-based Photo Sharing.

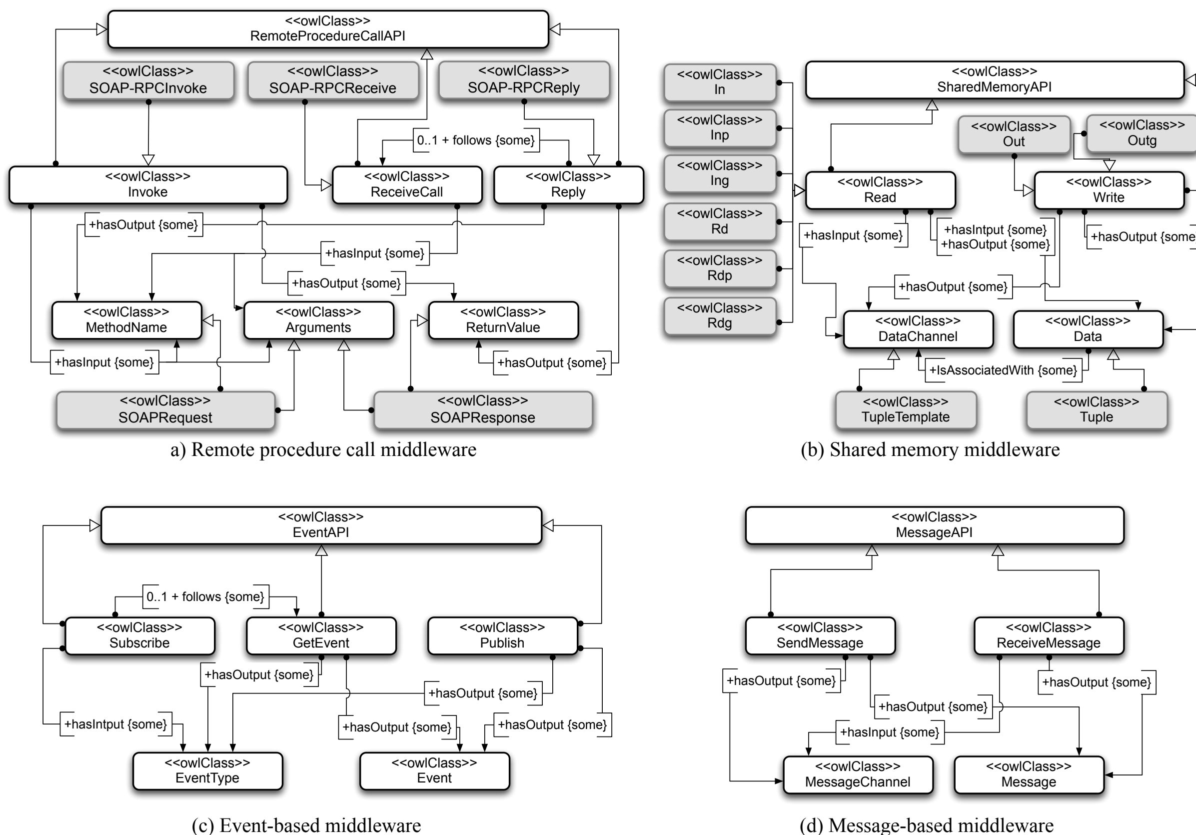
6.2 Ontology for Mediation

Towards enabling mediators, we introduce a middleware ontology that forms the basis of middleware protocol mediation (Section 6.2.1). In addition, domain-specific application ontologies characterizing application actions serve defining both control- and data-centric concepts (Section 6.2.2).

6.2.1 Middleware Ontology

In this section we propose: (i) *a reference middleware ontology* and (ii) a *middleware alignment*. The aim of the first is to provide a reference characterization of the existing middleware so to have a shared language to express middleware functions.
Instead, the middleware alignment, when applied to LTSs, realizes an abstraction of the middleware functions producing the corresponding LTSs with only application actions and input/output data. Towards the realization of the AMAzING process, (i) and (ii) serves realizing the middleware abstraction next to the application abstraction illustrated in the previous chapter. In more detail, middleware-specific functions are first abstracted as reference middleware functions of the reference ontology, and then aligned.
State-of-the-art middleware may be categorized according to four *middleware types* regarding provided communication and coordination services [116]: *remote procedure call, shared memory, event-based* and *message-based*.
The proposed **reference middleware ontology** is depicted in Figure 6.2 and more specifically with concepts defined in white boxes. The ontology is structured around four categories, which serve as reference enabling to express in a unique language (the set of white boxes concepts) functions of different middleware solutions.
Indeed, specific middleware functions (gray boxes in Figure 6.2) refines the reference middleware ontology functions (white boxes in Figure 6.2). In particular grayed boxes define concepts of the LIME and SOAP-based middleware solutions, that we consider in our Photo Sharing scenario, respectively refining concepts of the shared-memory and remote procedure call middleware reference ontology.

In addition to the *is-a* relation that is denoted by a white arrow head, 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[6] as illustrated by the *follows* relation that serves defining sequence patterns.



a) Remote procedure call middleware

(b) Shared memory middleware

(c) Event-based middleware

(d) Message-based middleware

Figure 6.2: Middleware ontology

The ontology is given as a set of UML diagrams. In Figure 6.2.a), the ontology concepts associated with RPC-based middleware include the *Invoke* function parameterized by the method name and arguments, which is used on the client side. 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 6.2.b). In the latter, messages are exchanged using the *SendMessage* and *ReceiveMessage* functions parameterized by the actual message and related channel (see Figure 6.2.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 6.2.c).

[6] `http://ontologydesignpatterns.org`

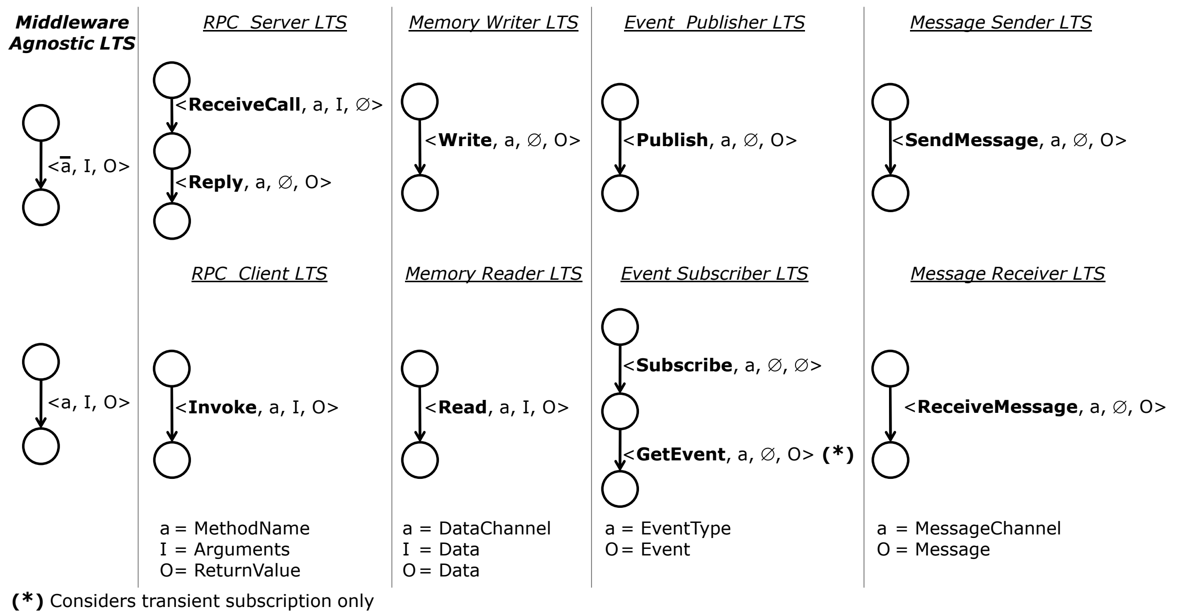


Figure 6.3: Middleware alignment

The proposed ontology serves mapping the functions of existing middleware into the reference functions, as illustrated for example for the cases of SOAP-based and LIME middleware. Heterogeneity in the underlying implementation may then be overcome using transparent middleware interoperability solutions (e.g., [25]).

A further challenge for connectors in pervasive networking environments is to enable mediation among different types of middleware. To enable such mediation, we introduce a **middleware alignment** that is 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 alignment 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 6.3. 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 *GetEvent* function, which is preceded by at least one invocation of *Subscribe* on the given event[7]. 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

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

action is defined by the execution of *ReceiveCall* followed by *Reply*, while the dual input action is defined by the *Invoke* function.

Figure 6.4: Shared-memory based Photo Sharing after the mapping of middleware functions to reference middleware ontology

As mentioned before, the alignments of Figure 6.3 are used to abstract protocols associated with the realization of affordances as *middleware-agnostic processes*. That is protocols where (sequence of) middleware functions are abstracted into input and output of the application actions. In other words middleware-agnostic processes have labels of the form $< a, I, O >$ where the application action a is overlined if the middleware function semantics is translated into output into the network while a is non-overlined if the middleware function semantics is translated into input from the network. As a result, protocols may be matched based purely on their application-specific actions. In more detail, middleware-specific functions are abstracted as middleware functions of the reference ontology, which are then translated into input and output actions through the defined alignment.

Figure 6.5: Middleware-agnostic peer-to-peer Photo Sharing

Figure 6.4 illustrates the protocol associated with the Shared-memory based Photo Sharing implementation where has been performed the mapping of its middleware-specific functions through the reference ontology. Instead, Figure 6.5 depicts the protocol associated with the peer-to-peer Photo Sharing implementation after the alignment into middleware-agnostic input and output application-specific actions. Following the previous

chapter, abstract processes are represented as Labeled Transition Systems where circles denote states (initial states are denoted by the double arrow and final states by double circles) and arrows denote transitions labeled by the corresponding 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.

6.2.2 Application-specific Ontology

The *subsumption* relation of ontologies (also called is-a) serves matching *application-specific actions*. We believe that the subsumption is not the panacea to reason about semantic relationships between concepts. Other relations such as sequence [42] or part-whole[8] should be specified. We believe that best practices of ontology design and ontology engineering[9] and the use of ontology design patterns[10] may prove very beneficial to automatically discover and reuse semantic relations between concepts. Indeed, next to the subsumption we defined other relations that serves matching the application-specific actions: *uses* and *is part of* relations.

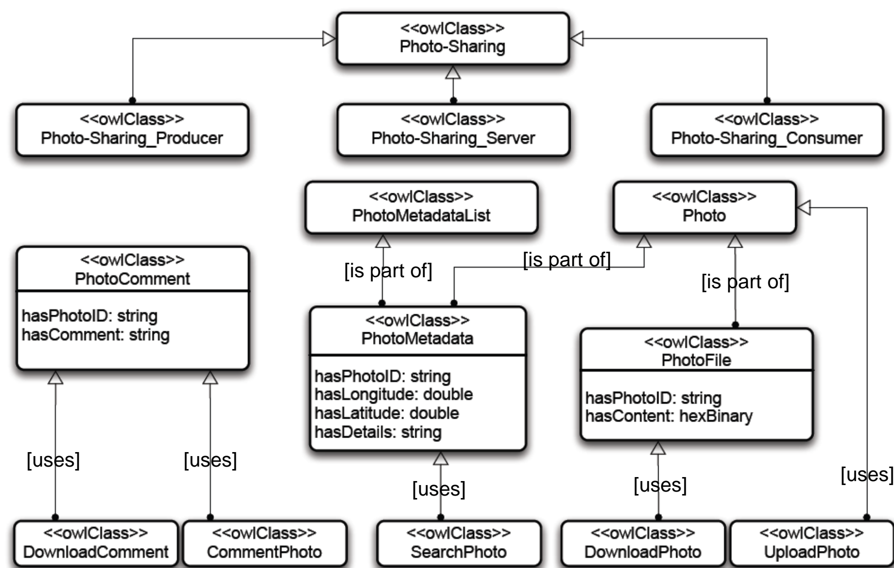


Figure 6.6: Photo Sharing ontology

As detailed in the next section, a required affordance (resp. input action) matches a provided affordance (resp. output action) if the former is subsumed by the latter.

[8] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html

[9] http://www.w3.org/2001/sw/BestPractices/OEP/

[10] http://ontologydesignpatterns.org

For illustration, Figure 6.6 gives an excerpt of the domain-specific ontology associated with our Photo Sharing scenario, which shows the relations holding among the various concepts implementing the scenario. 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.

6.3 Abstraction of Networked Systems

The above described models extensions leads to a refinement of the working of the abstraction step described in the previous chapter by manipulating models which include also middleware functions and input/output data. We recall, as described in Section 5.1.1, that the aim of this phase is to makes protocols comparable thus allowing to perform the functional matching check and possibly reduces their size so to ease the automatic reasoning on them.
We recall that, differently from before, here we first perform the affordance matching check illustrated in Section 6.4.1 to verify an high-level networked systems compatibility. If the check is successful, then we perform the abstraction otherwise the process stops because the networked systems are not compatible/composable. Within the abstraction (1) we add the middleware abstraction in the beginning. That is, we need to do a kind of preprocessing that translate the protocol model described here (including middleware functions, application actions and input/output data) into one which abstracts the middleware thus resulting in a protocol describing only application actions and input/output data. Subsequently, (2) we slightly modify the reasoning described in Section 5.1.1 to include also the data reasoning together with the application actions reasoning. While the middleware abstraction (1) is described in Section 6.2.1 we do not describe (2) which is similar to what already done for the application reasoning and is part of future work.

6.4 Functional Matching of Networked Systems

As already mentioned, we need to refine the working of the matching of the previous chapter according to the above provided characterization of networked systems and the described abstraction phase. In the following we provide such definition.

More precisely, given two networked systems implementing *semantically matching affordances* (Section 6.4.1), we say that the networked systems are **functionally matching** iff

there exists a *mapping of their interfaces* (Section 6.4.2) under which their *affordances are matching* (Section 6.4.3).
Then, considering two semantically matching affordances of two networked systems, we check the existence of an interface mapping that makes the abstracted affordance protocols match.
As already said, the semantic matching of affordances is introduced to limit the use of reasoning about behavioral matching, and hence improve the overall performance of the abstract mediator synthesis.

6.4.1 AFFORDANCE MATCHING

The first step in identifying the possible matching of two networked systems is to assess whether they respectively provide and require a matching affordance. More precisely, and following the definition of [91] that adheres to the Liskov Substitution Principle [74]:

an affordance $Aff_R = <$ *Req, Op_R, I_R, O_R* $>$ matches an affordance $Aff_P = <$*Prov, Op_P, I_P, O_P* $>$, denoted with $Aff_R \hookrightarrow Aff_P$, *iff* in the given ontology
Op_R $\sqsubseteq$ Op_P;
$\mathrm{I}_P \sqsubseteq \mathrm{I}_R$ which is a shorthand notation for subsumption between sets of ontology concepts;
$\mathrm{O}_R \sqsubseteq \mathrm{O}_P$, similar shorthand notation as above.

Note that an affordance Aff_R of type Req *produces* the *inputs* I_R and *consumes* the corresponding *outputs* O_R. In a dual manner, an affordance Aff_P of type $Prov$ *consumes* the *inputs* I_P and *produces* the corresponding *outputs* O_P.

In the case where an affordance is required and provided by a NS (i.e., $Type$ = *Req_Prov*) and the other affordance is required (resp. provides) by another NS, we apply similarly the condition above considering that the *Req_Prov* affordance is provided (resp. required). For instance, given (1) and (2) below we have (3):

```
PhotoSharingConsumer =<Req, Photo−Sharing_Consumer, Comment, Photo> (1)

PhotoSharing =<Req_Prov,Photo−Sharing,{Photo, PhotoComment},{Photo, PhotoComment}> (2)

PhotoSharingConsumer ↪ PhotoSharing (3)
```

Finally, in the case where both affordances are both provided and required, the equivalence of concepts obviously needs to hold.
Given affordance matching, the connector between the matching networked systems should mediate possible mismatches in their respective middleware-agnostic interaction protocols. Specifically, possible 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 affordance of interest. On the other hand, the absence of consumption of an output action does not affect the behaviour of the networked system as long as deadlock is prevented by the connector at runtime. Still, synthesis of

a protocol mediator is known as a computationally hard problem for finite state systems in general [29] and thus requires heuristics to make the problem tractable. Towards that goal, we focus on enabling the basic mediation patterns described in Chapter 4 [109, 108]. We then account for basic mediation patterns as follows:

- **Messages ordering pattern:** (includes ordering mismatch) concerns the re-ordering of actions so that networked systems may indeed coordinate. Assuming the specification of affordance behaviour using a BPEL-like language as discussed in Section 6.1.3, causally independent actions may be identified through data-flow analysis, hence enabling to introduce concurrency among actions and thus supporting acceptable re-ordering.
- **Message consumer pattern:** (includes Extra send/missing receive mismatch) as discussed above, extra output actions are simply discarded from the standpoint of behavioural matching. Obviously, the associated mediator should handle any extra synchronous output action to avoid deadlock.
- **Message producer pattern:** (includes Missing send/extra receive mismatch) 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.
- **Message Splitting pattern:** (includes One send-many receive/many receive-one send mismatch) 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.
- **Message Merger pattern** (includes Many send-one receive/one receive-many send mismatch) 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.
- **Message Translator Pattern:** (which includes Signature mismatch) concerns the language translation among actions so that NSs can coordinate. While doing the behavioural matching, there is the assumption of the middleware ontology and of the application-specific ontology which allow alignment. Then, the mediator should just exploit that information.

6.4.2 Interface Mapping

Following the above, interface mapping serves identifying mappings among the actions of the interaction protocols run by the networked systems that should coordinate (to reach a common goal) towards the realization of a given affordance.

Let two networked systems that respectively implement the matching affordances Aff_1 and Aff_2. Let further $\mathcal{I}_{Aff_1}$ (resp. $\mathcal{I}_{Aff_2}$) be the set of abstracted actions (of the form application action, input data, output data) executed by the protocol realizing *Aff*$_1$ (resp. *Aff*$_2$); We introduce the function $Map_I(\mathcal{I}_{Aff_1}, \mathcal{I}_{Aff_2})$ which identifies the set of all possible mappings of all the input actions of I_{Aff_1} (resp. I_{Aff_2}) with actions of $\mathcal{I}_{Aff_2}$ (resp. $\mathcal{I}_{Aff_1}$), according to the semantics of actions and data. More formally:

$$\begin{gathered} Map_I(\mathcal{I}_{Aff_1}, \mathcal{I}_{Aff_2}) = \\ (\bigcup_{<a,I,O> \in \mathcal{I}_{Aff_1}} map(<a, I, O>, \mathcal{I}_{Aff_2})) \quad \cup \\ (\bigcup_{<a',I',O'> \in \mathcal{I}_{Aff_2}} map(<a', I', O'>, \mathcal{I}_{Aff_1})) \end{gathered}$$

where:

$$\begin{gathered} map(<a, I_a, O_a>, \mathcal{I}) = \\ (\bigcup_{i=1..n} \; <\bar{b}_i, I_i, O_i> \in \mathcal{I} \;\; | \;\; a \sqsubseteq \bigcup_i \{b_i\} \quad \wedge \quad I_i \sqsubseteq (\bigcup_{j<i} O_j) \cup \{I_a\} \quad \wedge \\ O_a \sqsubseteq (\cup_{j<i} \{O_j\}) \cup \{I_a\}) \quad \wedge \\ \forall seq_1 \in map(<a, I_a, O_a>, \mathcal{I}), \nexists seq_2 \in map(<a, I_a, O_a>, \mathcal{I}) | seq_2 \prec seq_1 \end{gathered}$$

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 affordance. 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 in Listing 6.3 the interface mapping between the *PhotoSharingConsumer* and *PhotoSharing* affordances. 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.

```
1 Map( I'_photo_sharing_consumer , I'_photo_sharing ) = {
2       < SearchPhotos, photoMetadata, photoMetadataList > ↦ {<<PhotoMetadata‾, ∅, photoMetadata >>} ,
3       < DownloadPhoto, photoID, photoFile > ↦ {<< PhotoFile‾ , ∅, photoFile >>} ,
4       < CommentPhoto, photoComment, acknowledgment > ↦ {<< PhotoComment‾, ∅, photoComment >>} ,
5       < DownloadComment, photoID, photoComment > ↦ {<< PhotoComment‾, ∅, photoComment >>} ,
6       < PhotoComment, photoID, photoComment > ↦ ∅ ,
7       < PhotoMetadata, photoMetadata, photoMetadataList > ↦ ∅ ,
8       < PhotoFile, photoID, photoFile > ↦ ∅
9 }
```

Listing 6.3: Interface mapping between the *PhotoSharingConsumer* and *PhotoSharing* affordances

We are currently devising an efficient algorithm implementation towards computing interface mappings, building upon related effort in the area (e.g., [81]). In addition, we

have implemented more primitive interface mapping that deals only with 1-to-1 action mapping. This allows for joint implementation of interface mapping and of behavioral matching of affordances using ontology-based model checking [16].

6.4.3 Behavioural Matching of Affordances

Given interface mappings returned by Map_I, we need to identify whether the protocols associated with the matching affordances may indeed coordinate reaching a common goal, i.e., the concurrent execution of the two protocols successfully terminates.

We say that a process P_1 associated with affordance Aff_1 *behaviourally matches* a process P_2 associated with affordance Aff_2 under $Map(\mathcal{I}_1', \mathcal{I}_2')$, denoted with $P_1 \hookrightarrow_{\mathcal{P}} P_2$, iff there exists at least one pair of complementary coordination policies.

Applying the above definition to our Photo Sharing example, we can check that:
$P_{photo_sharing_consumer} \hookrightarrow_{\mathcal{P}} P_{photo_sharing}$

6.5 Mapping of Networked Systems

According to the refined models, abstraction, and matching proposed in this chapter, we also need to refine the working of the mapping or synthesis of mediators. In particular, from a theoretical point of view we can leverage on the definition of the previous chapter extended with the data. While special care is required from a pragmatic standpoint, in reversing the abstractions that are applied to be able to reason about interface mapping and protocol matching. In other words, the knowledge embedded in the ontologies needs to be used to both abstract and concretize concepts. For instance, a middleware agnostic action needs to be ultimately translated into a middleware-specific message that embeds application-specific control and data. We are currently investigating such an issue, where the encoding of knowledge using domain specific languages seems very promising [26].

6.6 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 chapter has discussed the foundations of mediators, 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 connectors specifically lies in the appropriate modeling of the networked systems' high-level functionalities, for which we exploit ontologies, and related protocols. This chapter has illustrated the refinement of our theory enabling mediators that in the same time address interoperability at both application- and middleware-layer. This represents our contribution compared to related work, that deals with either automated protocol conversion/mediation or middleware interoperability. In particular, through the alignment of middleware concepts, we are able to deal with interoperability between networked systems relying on heterogeneous middleware paradigms. As future work we need to further formalize some of the above described refinement, as for instance the working of abstraction and mapping phases.
Further we plan to implement the matching and mapping relations discussed in this chapter so to enable networked systems to actually meet and compose on the fly. This means that we need to design and implement the networked systems matching and mapping as part of a networked systems' universal discovery.

CHAPTER 7

MEDIATING FLICKR AND PICASA: A CASE STUDY

In the previous chapters we illustrated a theory of mediating connectors to achieve interoperability exploiting a running scenario, the photo sharing within a stadium (see also [15, 62, 43]). Other analyzed scenarios, such as Instant Messaging Applications and Distributed Marketplace Scenario, can be found in our works [110, 108, 107, 109, 18].

In order to validate our theory and process, together with the Instant Messengers Applications, we analyzed another *real world case study*, i.e., the Photo Management and Sharing Application Tools which we describe in the following.

We considered two real services **Flickr** [50] and **Picasa** [93], illustrated in Figure 7.1, offering similar and compatible functionalities.

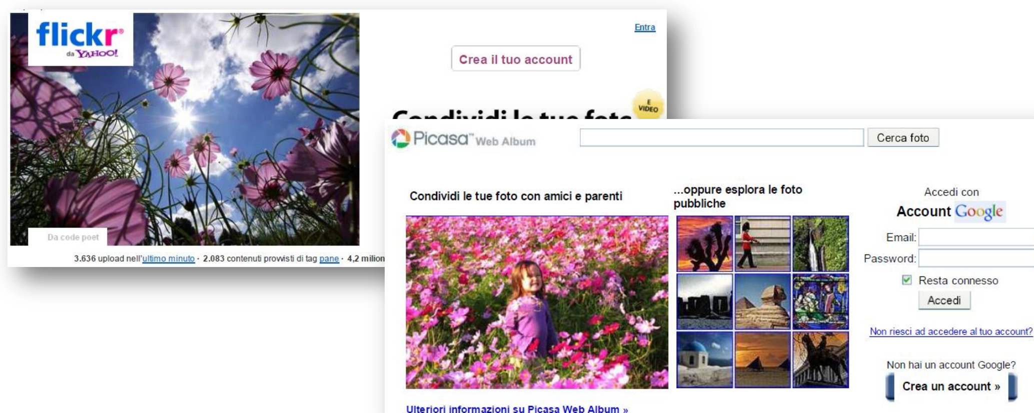


Figure 7.1: Flickr and Picasa services

Both of them provide an API (see [51] for Flickr and [94] for Picasa) which we exploited for the implementation; Picasa also offers a java example useful to developers [95]. For Flickr, more specifically, we used the REST [49] access to the service.

The above mentioned tools, among others functionalities, (i) allow users to upload their pictures on the web making available a space to them where to store the photos, and (ii) share the pictures with, e.g., family, friends, colleagues. They are *functionally matching*

protocols while having some *behavioural mismatches* indeed they offer similar operations while showing some discrepancies in their implementation.

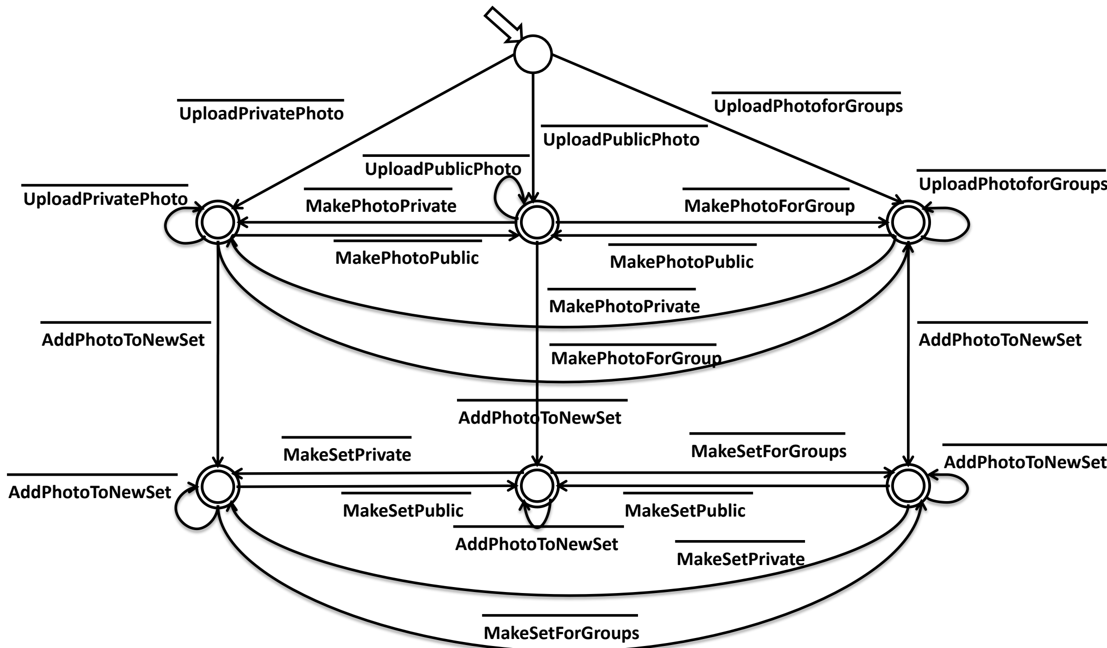


Figure 7.2: Flickr client protocol

In particular, we focused on a Flickr client and the Picasa server. Figures 7.2 and 7.3 illustrate the considered protocols respectively. The labels on the LTSs are self-explanatory. A different version of this example has been used in [31] where the authors describe an approach on how to infer mappings without or with limited human intervention.

A first difference between the Flickr and Picasa protocols concerns the operations signatures of the two services as can be noticed from their labels. Then, in Flickr, photos can have different levels of visibility spanning public, private, family, and friend, where family and friend are groups and the meaning is that only such authorized people can see the photos. Private and public visibility are self-explanatory. Once uploaded, the pictures can be organized in sets and it is always possible to modify the photo visibility and that of the sets.
In Picasa, instead, the albums play a central role and hence the tool does not support the upload of non-grouped photos, i.e. photos not organized in album. For this reason in Picasa is needed first to create an album in order to be able to then upload pictures into the created album. Moreover, due to this centrality of albums, differently from Flickr, the visibility can be only applied to albums and not directly to photos. Additionally, the visibility in Picasa can be private, public, anyone with the link and additionally can be also shared with family, friends and colleagues (which are groups).

In the following we describe the application our `MediatorS` theory based on the `AMAzING` process on these two protocols. As already said, we assume to have the two protocols together their ontological information. Figure 7.4 shows the ontological information related

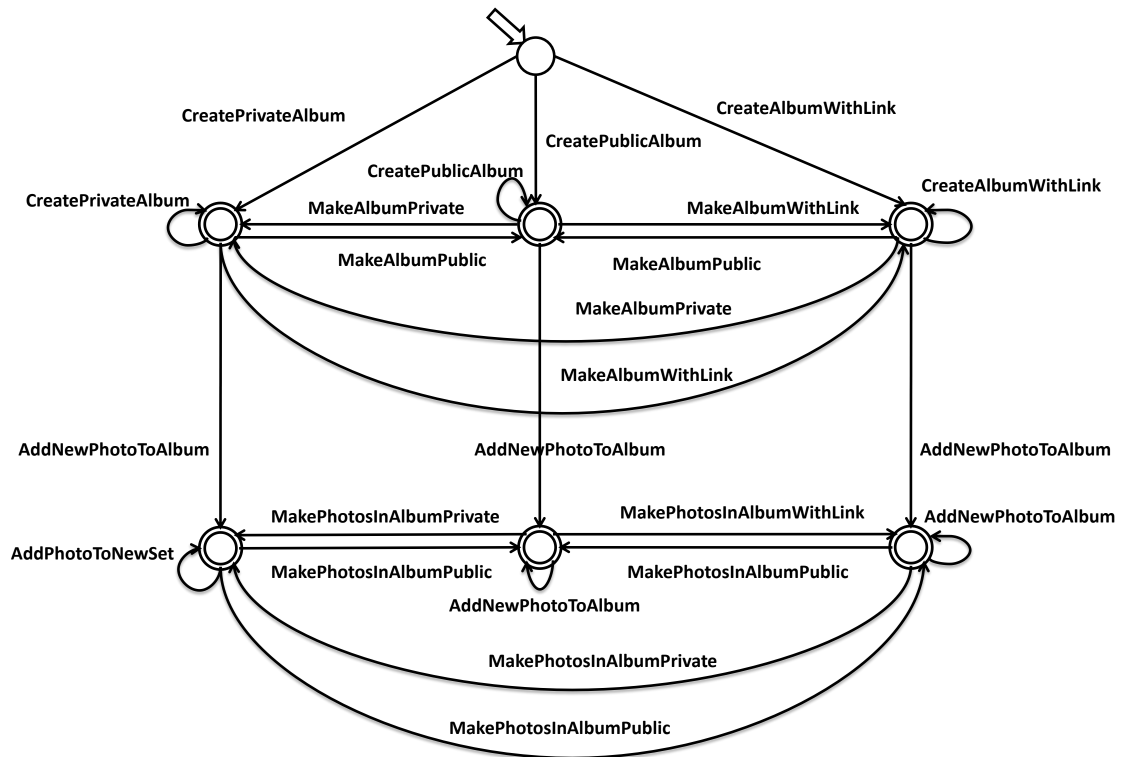


Figure 7.3: Picasa server protocol

to Flickr (left column) and Picasa (right column) protocols. The central column shows their common language where in some cases there is no correspondence meaning that those actions have not any correspondent action.

The first step is the protocols **abstraction** by using the common language. The abstracted protocols for this scenario are illustrated in Figure 7.5 where it appears clear that they are the "same protocol". The abstraction information is made up by the ontology mapping rules used in order to produce the abstracted LTS. For instance $\overline{PuAPuP}$ in Figure 7.5 a) can be obtained in several ways and we store all the possible mappings. Examples are: $\overline{UploadPrivatePhoto}$. $\overline{AddPhotoToNewSet}$. $\overline{MakeSetPublic} \mapsto \overline{PuAPuP}$; $\overline{UploadPublicPhoto}$. $\overline{AddPhotoToNewSet} \mapsto \overline{PuAPuP}$.
Referring to Figure 7.5 b), $PuAPuP$ can be obtained in several ways. Examples are:
$CreatePublicAlbum$. $AddNewPhotoToAlbum \mapsto PuAPuP$;
$CreatePrivateAlbum$. $AddNewPhotoToAlbum$. $MakePhotosInAlbumPublic \mapsto PuAPuP$.

The second step is the **matching**. In our case study, Flickr and Picasa protocols are compatible having all compatible (abstract) traces, i.e. their abstract protocols are the same (complementary) protocol. The intuitive meaning of the matching traces is that there are three high-level matching functionalities: (1) upload of public albums with public pictures, (2) upload of private albums with private pictures, and (3) upload of albums only visible to some group with pictures only visible to some group.

Flickr Client Protocol	Common Language Projected on the Protocols		Picasa Server Protocol
$\overline{\text{UploadPrivatePhoto}}$. $\overline{\text{AddPhotoToNewSet}}$. $\overline{\text{MakeSetPublic}}$	$\overline{PuAPuP}$	$PuAPuP$	CreatePublicAlbum. AddNewPhotoToAlbum
$\overline{\text{UploadPrivatePhoto}}$. $\overline{\text{AddPhotoToNewSet}}$. $\overline{\text{MakeSetPublic}}$	$\overline{PuAPuP}$	$PuAPuP$	CreatePrivateAlbum. AddNewPhotoToAlbum. MakePhotosInAlbumPublic
$\overline{\text{UploadPublicPhoto}}$. AddPhotoToNewSet	$\overline{PuAPuP}$	$PuAPuP$	CreatePublicAlbum. AddNewPhotoToAlbum
⋮	⋮	⋮	⋮
$\overline{\text{UploadPrivatePhoto}}$	$\overline{PrP}$	-	-
$\overline{\text{AddPhotoToNewSet}}$	$\overline{PrS}$	-	-
$\overline{\text{MakeSetPublic}}$	$\overline{PuS}$	-	-
-	-	PrA	CreatePrivateAlbum
-	-	$PrPA$	AddNewPhotoToAlbum
-	-	PuA	MakePhotosInAlbumPublic
$\overline{\text{UploadPrivatePhoto}}$. $\overline{\text{AddPhotoToNewSet}}$	$\overline{PrAPrP}$	$PrAPrP$	CreatePrivateAlbum. AddNewPhotoToAlbum.
$\overline{\text{UploadPhotoforGroups}}$. AddPhotoToNewSet	$\overline{GrAGrP}$	$GrAGrP$	CreateAlbumWithLink. AddNewPhotoToAlbum
⋮	⋮	⋮	⋮

Figure 7.4: Ontological information

We recall that mapping back each abstract trace into concrete traces of Flickr and Picasa protocols we obtain a set of different traces, i.e., a protocol. Hence we store as matching information the pairs of matching traces of the two protocols.

The third and last step is the **mapping** that, taking in input the abstraction and the matching information, compute the mediator.
The mediator resulting from this step intuitively translates any service invocation coming from the Flickr client into the corresponding service invocation to the Picasa server. More in detail, the mediator receives *any trace* of Flickr (corresponding to one of the three high-level functionalities (1),(2),or (3) above) and translates them into the *set of complementary traces* of Picasa.
Figure 7.6 shows the mediator between Flickr and Picasa (from the former to the latter) where for the readability of the figure we relabeled the actions with capital letters labels. The upper part of the figure, coloured in blue, is the complementary protocol of the Flickr client, i.e., the same protocol with opposite actions (all actions are of the type receive). The bottom part of the figure, coloured in green, is made up by three copies of the Picasa client protocol, i.e. protocol with all send actions. This is due to the fact that from the matching check among Flickr and Picasa protocols we discover that there are three com-

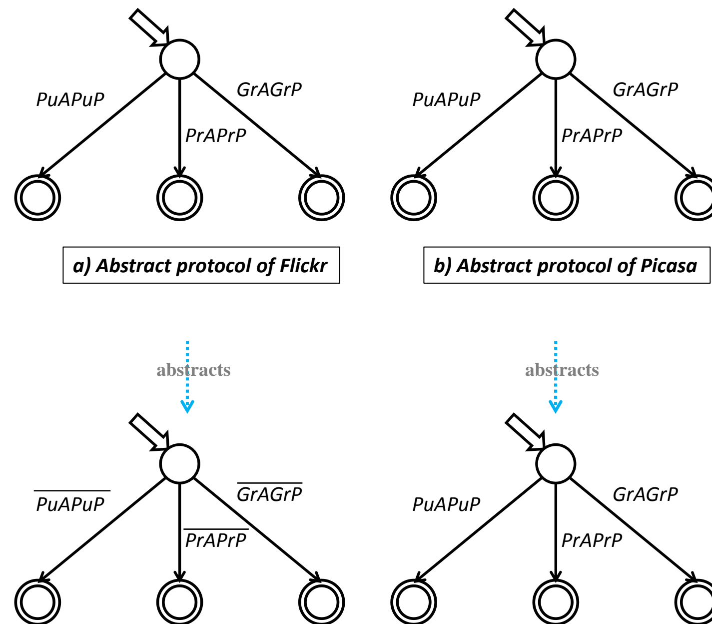


Figure 7.5: Abstracted protocols

plementary functionalities and each of them can be achieved in several different ways. For this reason we can see that the mediator (Figure 7.6) first accepts the input coming from Picasa (in the upper portion of the figure) and then depending on which functionality has been performed in Picasa will perform one of its corresponding behaviour in Flickr terminating in one of the three accepting states in the bottom of the figure. In particular a correct interaction will be represented in the mediator as a trace execution that starts from the initial state and ends into one of the three final states.

We have also implemented a mediator to allow the communication between Flickr and Picasa. The implementation has been done using Java and exploiting the API provided by the two services (mentioned in the beginning of this chapter). Two personal accounts on the photo management and sharing application tools have been used.

In particular we considered the Flickr client first uploading a private photo, subsequently adding the photo to a new private set, and then making public the visibility of the set. This amount to uploading a public set with a public photo on it. Hence this can be translated into one of the corresponding invocations of the Picasa service to do that.
The case described above is illustrated by the following coordinations policies from Flickr and Picasa clients respectively: $\overline{UploadPrivatePhoto}$. $\overline{AddPhotoToNewSet}$. $\overline{MakeSetPublic}$ and $\overline{UploadPublicPhoto}$. $\overline{AddPhotoToNewSet}$. As we said, the mediator (a) receives the service invocations from the Flickr client (expressed in its language and protocol), i.e., $UploadPrivatePhoto$. $AddPhotoToNewSet$. $MakeSetPublic$ (b) translate them into a Picasa service invocation, i.e., $\overline{UploadPublicPhoto}$. $\overline{AddPhotoToNewSet}$, and (c) invoke the Picasa service.
Summarizing, the resulting mediator trace is $UploadPrivatePhoto$. $AddPhotoToNewSet$

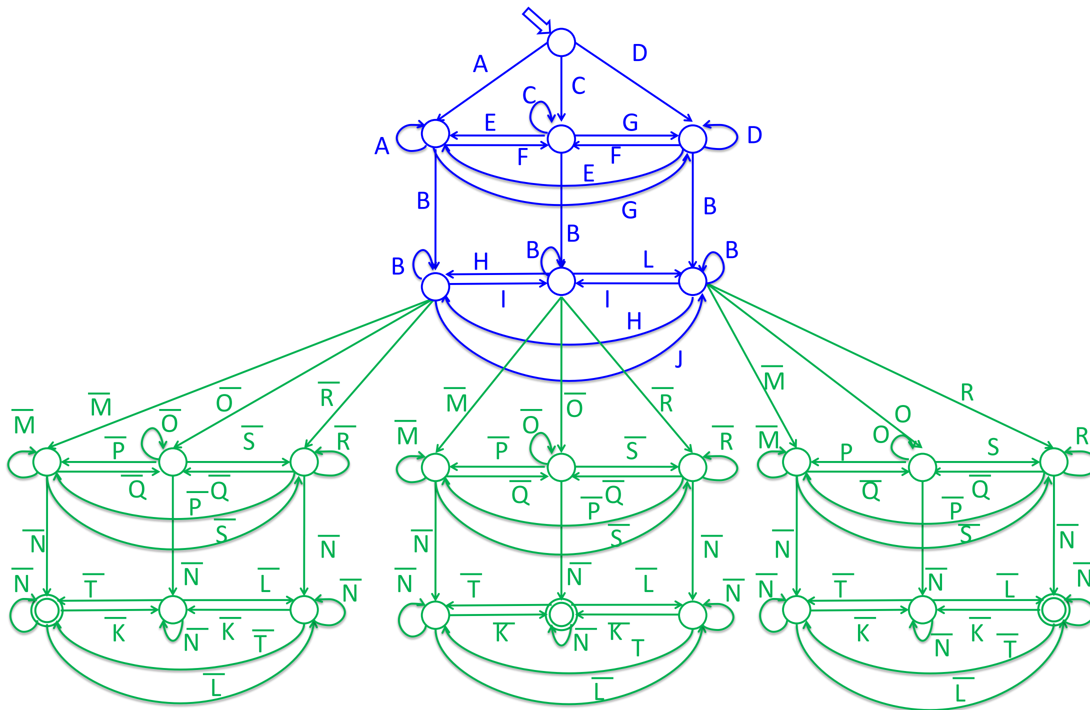


Figure 7.6: Mediator between Flickr and Picasa protocols

. $MakeSetPublic$. $\overline{UploadPublicPhoto}$. $\overline{AddPhotoToNewSet}$. The outcome of this interaction is that the Flickr client perform its invocations to upload the public set with a public photo, the mediator intercepts these invocations and translate them into the corresponding invocations of Picasa service then creating the album with the photo in Picasa.

This case study have shown that both the process and the theory are successfully being able to: (1) establish that Flickr and Picasa clients are functionally matching, and (2) produce a mediator that lets a Flickr client interact/communicate with the Picasa service. This can be considered a first effort towards the automated synthesis of mediators. Indeed, to put our solution into actual practice, we still need to implement the reasoning underlying the `MediatorS` theory.

CHAPTER 8

THE MEDIATORS THEORY IN CONNECT

The CONNECT[1] Integrated Project [35, 64] aims at enabling continuous composition of Networked Systems (NSs) to respond to the evolution of functionalities provided to and required from the networked environment.
At present the efficacy of integrating and composing networked systems depends on the level of interoperability of the systems's underlying technologies. However, interoperable middleware cannot cover the ever growing heterogeneity dimensions of the networked environment.
CONNECT aims at dropping the interoperability barrier by adopting a revolutionary approach to the seamless networking of digital systems, that is, synthesizing on the fly the connectors via which networked systems communicate.
The resulting emergent connectors are effectively synthesized according to the behavioural semantics of application- down to middleware-layer protocols run by the interacting parties. The synthesis process is based on a formal foundation for connectors, which allows learning, reasoning about and adapting the interaction behaviour of networked systems at run-time. Synthesized connectors are concrete emergent system entities that are dependable, unobtrusive, and evolvable, while not compromising the quality of software applications.
To reach these objectives the CONNECT project undertakes interdisciplinary research in the areas of behaviour learning, formal methods, semantic services, software engineering, dependability, and middleware.
Specifically, CONNECT will investigate the following issues and related challenges: (i) Modeling and reasoning about peer system functionalities, (ii) Modeling and reasoning about connector behaviours, (iii) Runtime synthesis of connectors, (iv) Learning connector behaviours, (v) Dependability assurance, and (vi) System architecture. The effectiveness of CONNECT research will be assessed by experimenting in the field of wide area, highly heterogeneous systems where today's solutions to interoperability already fall short (e.g., systems of systems).

Due to its research topics and objectives, CONNECT *adopted and currently exploits both the* `AMAzING` *approach and the* `MediatorS` *theory*. In particular, given the respective interaction behaviour of networked systems (NSs), CONNECT wants to synthesize the behaviour of the connector(s) needed for them to interact. These connectors serve as

[1] http://connect-forever.eu/

mediators of the networked systems interaction at both application and middleware layers. First achievements have been reported in Deliverable D3.1 [38]: Modeling of application- and middlewarelayer interaction protocols while more recent results are contained in [39]. More specifically, the project used AMAzING and MediatorS to address two among all its objectives that can be summarized as follows:

1 - Synthesis of application-layer conversation protocols. The goal here is to identify connectors patterns that allow the definition of methodologies to automatically synthesize, in a compositional way and at run-time, application-layer connectors.

2 - Synthesis of middleware-layer protocols. The objective here is to generate adequate protocol translators (mappings) that enable heterogeneous middleware to interoperate, and realize the required non-functional properties, thus successfully interconnecting NSs at the middleware level.

Thus, the project has been and is a perfect framework within which to validate and experiment MediatorS and the process it implements.
To give an overview of the whole CONNECT, in the following we report first results towards its architecture. More details of preliminary results can be found in [15, 36] while more recent results are in [37].

8.1 TOWARDS THE CONNECT ARCHITECTURE

A fundamental requirement of distributed systems is to ensure interoperability between the communicating elements; systems that have been implemented independently of one another must be able to connect, understand and exchange data with one another. This is particularly true in highly dynamic application domains (e.g. mobile and pervasive computing) where systems typically only encounter one another at runtime. Middleware technologies have traditionally resolved many of the interoperability problems arising in these situations, such as operating system and programming language heterogeneity. Where two applications conform to a particular middleware standard, e.g. CORBA [57] and Web Services [20, 33], they are guaranteed to interoperate. However, the next generation of distributed computing applications are characterized by two important properties that force a rethink of how interoperability problems should be tackled:

- Extreme heterogeneity. Complex pervasive systems are composed of technology-dependent islands, i.e. domain specific systems that employ heterogeneous communication and middleware protocols. For example, Grid applications, mobile ad-hoc networks, Enterprise systems, and sensor networks all use their own protocols such that they cannot interoperate with one another.

- Spontaneous Communication. Connections between systems are not made until runtime (and are made between systems that were not aware of one another beforehand). With such characteristics, requiring all applications to be developed upon a common middleware technology, e.g. CORBA or Web Services, is unsuitable in practice. Rather, new approaches are required that allow systems developed upon heterogeneous technologies to interoperate with one another at runtime. In this paper, we present the CONNECT architectural framework that aims to resolve this interoperability challenge in a fundamentally different way. Rather than create a middleware solution that is destined to be yet another legacy platform that adds to the interoperability problem, we propose the novel approach of generating the required middleware at runtime i.e. we synthesize the necessary software to connect two end-systems. For example, if a client application developed using SOAP [58] encounters a CORBA server then the framework generates a connector that resolves the heterogeneity of the i) data exchanged, ii) application behaviour e.g. sequence of operations called, and iii) the lower level middleware and network communication protocols. In this paper we identify the requirements that need to be satisfied to guarantee interoperability, namely interoperability at the discovery, behavioural and data level. We then outline the key elements of the CONNECT framework that underpin a runtime solution to achieving such interoperability, and that are further detailed in the papers [60, 62, 18, 6, 41]:

- Discovering the functionality of networked systems and applications advertised by legacy discovery protocols e.g. Service Location Protocol (SLP) and Simple Service Discovery Protocol (SSDP). Then, transforming this to a rich intermediary description used to syntactically and semantically match heterogeneous services.

- Using learning algorithms to dynamically determine the interaction behaviour of a networked system from its intermediary representation and producing a model of this behaviour in the form of a labelled transition system (LTS) [60].

- Dynamically synthesising a software mediator using code generation techniques (from the independent LTS models of each system) that will connect and co-ordinate the interoperability between heterogeneous end systems [62, 18].

We highlight the potential of this CONNECT framework to achieve interoperability having high levels of heterogeneity.

8.1.1 HETEROGENEITY DIMENSIONS

We now examine the dimensions of heterogeneity which explain why interoperation fails.

1. Heterogeneous discovery protocols are used by systems to locate other systems, and to advertise their services, e.g., SLP and SSDP. In situations where systems differ in this aspect, they will be unable to discover one another and the first step fails.

2. Systems use heterogeneous middleware protocols to implement their functional interactions, e.g. tuple space middleware and SOAP RPC protocol [58] are used; these are different communication paradigms: the tuple space follows a shared space abstraction to write tuples to and read from, whereas RPC is an asynchronous invocation of a remote operation. Hence, the two cannot interoperate directly.

3. Application level heterogeneity. Interoperability challenges at the application level arise due to the different ways that application developers implement the functionality. As a specific example, consider two different sequences of messages: a single remote call, or three separate remote calls.

4. Data-representation Heterogeneity. Implementations may represent data differently. Data representation heterogeneity is typically manifested at two levels. The simplest form of data interoperability is at the syntactic level where two different systems may use very different formats to express the same information. As example consider an XML representation vs. a Java Object. Further, even if two systems share a common language to express data, different dialects may still raise interoperability issues, e.g. price and cost or also value and amount. The deeper problem of data heterogeneity is the semantic interoperability whereby all systems should have the same interpretation of data.

Summarizing the requirements, we have described four dimensions where systems may be heterogeneous: i) the discovery protocol, ii) the middleware protocol, iii) application behaviour, and iv) data representation and meaning. A universal interoperability solution must consider all four.

8.1.2 Beyond State of the Art Interoperability Solutions

Achieving interoperability between independently developed systems has been one of the fundamental goals of middleware researchers and developers; and prior efforts have largely concentrated on solutions where conformance to one or other standard is required e.g. as illustrated by the significant standards work produced by the OMG for CORBA middleware [57], and by the W3C for Web Services based middleware [20, 33]. These attempt to make the world conform to a common standard; such an approach has been effective in many areas e.g. routing of network messages in the Internet. To some extent the two approaches have been successful in connecting systems in Enterprise applications to handle hardware platform, operating system and programming language heterogeneity. However, in the more general sense of achieving universal interoperability and dynamic interoperability between spontaneous communicating systems they have failed. Within the field of distributed software systems, any approach that assumes a common middleware or standard is destined to fail due to the following reasons:

- A one size fits all standard/middleware cannot cope with the extreme heterogeneity of distributed systems e.g. from small scale sensor applications through to large scale Inter-

net applications.

- New distributed systems and application emerge fast, while standards development is a slow, incremental process. Hence, it is likely that new technologies will appear that will make a pre-existing interoperability standard obsolete, c.f. CORBA versus Web Services (neither can talk to the other).

- Legacy platforms remain useful. Indeed, CORBA applications remain widely in use today. However, new standards do not typically embrace this legacy issue; this in turn leads to immediate interoperability problems. One approach to resolving the heterogeneity of middleware solutions comes in the form of interoperability platforms. ReMMoC [56], Universal Interoperable Core [100] and WSIF [44] are client side middleware which employ similar patterns to increase interoperability with heterogeneous service side protocols. First, the interoperability platform presents an API for developing applications with. Secondly, it provides a substitution mechanism where the implementation of the protocol to be translated to, is deployed locally by the middleware to allow communication directly with the legacy peers (which are simply legacy applications and their middleware). Thirdly, the API calls are translated to the substituted middleware protocol. For the particular use case, where you want a client application to interoperate with everyone else, interoperability platforms are a powerful approach. However, these solutions rely upon a design time choice to develop upon the interoperability platforms. Therefore, they are unsuited to other interoperability cases e.g. when two applications developed upon different legacy middleware want to interoperate spontaneously at runtime. Software bridges offer another interoperability solution to enable communication between different middleware environments. Clients in one middleware domain can interoperate with servers in another middleware domain where the bridge acts as a one-to-one mapping between domains; it will take messages from a client in one format and then marshal this to the format of the server middleware; the response is then mapped to the original message format. While a recognised solution to interoperability, bridging is infeasible in the long term as the number of middleware systems grow i.e. due to the effort required to build direct bridges between all of them. The Enterprise Service Buses (ESB) can be seen as a special type of software bridge; they specify a service-oriented middleware with a message-oriented abstraction layer atop different messaging protocols (e.g., SOAP, JMS, SMTP). Rather than provide a direct one-to-one mapping between two messaging protocols, a service bus offers an intermediary message bus. Each service (e.g. a legacy database, JMS queue, Web Service etc.) maps its own message onto the bus using a piece of code, to connect and map, deployed on the peer device. The bus then transmits the intermediary messages to the corresponding endpoints that reverse the translation from the intermediary to the local message type. Hence traditional bridges offer a 1-1 mapping; ESBs offer an N-1-M mapping. Example ESBs are Artix [1] and IBM Websphere Message Broker [2]. ESBs offer a solution to the problem of middleware heterogeneity; however, it focuses on the messaging abstraction only and the assumption is that all messaging services can be mapped to the intermediary abstraction (which is a general subset of messaging protocols). This

decision is enacted at design or deployment time, as the endpoint must deploy code to connect to a particular message bus with an appropriate translator and hence is unsuitable for dynamic interoperation between two legacy platforms. INDISS [25], uMiddle [83], OSDA [73], PKUAS [55] and SeDiM [52] are examples of transparent interoperability solutions which attempt to ensure legacy solutions unaware of the heterogeneous middleware are still able to interoperate. Here, protocol specific messages, behaviour and data are captured by the interoperability framework and then translated to an intermediary; a subsequent mapper then translates from the intermediary to the specific legacy middleware to interoperate with. The use of an intermediary means that one middleware can be mapped to any other by developing these two elements only (i.e. a direct mapping to every other protocol is not required). Another difference to bridging is that the peers are unaware of the translators (and no software is required to connect to them, as opposed to connecting to 'bridges'). The interoperation solutions proposed above concentrate on the middleware level. They support interoperation by abstract protocols and language specifications. But, by and large they ignore the data dimension. To this extent a number of efforts, which are generically labelled as Semantic Web Services [77, 99, 47], attempt to enrich the Web services description languages with a description of the semantics of the data exchanged. The result of these efforts are a set of languages that describe both the orchestration of the services' operations, in the sense of the possible sequences of messages that the services can exchange as well as the meanings of these messages with respect to some reference ontology. However, such approaches assume a common middleware standard and do not address the heterogeneity problems previously described. The state of the art investigation shows two important things; first, there is a clear disconnect between the main stream middleware work and the work on application, data, and semantic interoperability; second, none of the current solutions addresses all of the four requirements of dynamic pervasive systems as highlighted previously. Hence, these results show that there is significant potential for CONNECT to extend beyond the state of the art in interoperability middleware.

8.1.3 THE CONNECT ARCHITECTURAL FRAMEWORK

The CONNECT architecture provides the underlying principles and software architecture framework to enact the necessary mechanisms to achieve universal interoperability between heterogeneous systems.

Figure 8.1 presents a high-level overview of the following actors involved within the CONNECT architecture and how they interact with one another:

- Networked systems are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- Enablers are networked entities that incorporate all the intelligence and logic offered by CONNECT for enabling connection between heterogeneous networked

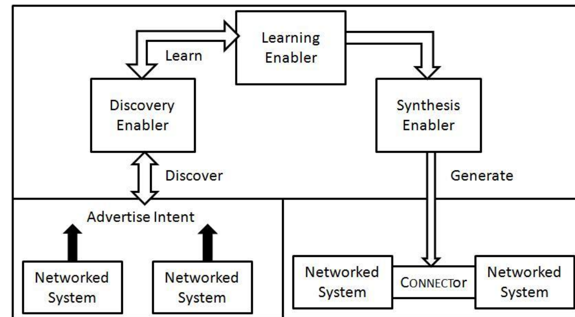


Figure 8.1: Actors in the CONNECT Architecture

systems. In this paper, we focus on how the discovery, learning and synthesis enablers co-ordinate to produce a connector as shown in Figure 8.1.

- Connectors are the synthesized software connectors produced by the action of enablers to connect networked systems.

Discovery and Learning of Networked Systems Networked systems use discovery protocols to advertise their will to connect (i.e. their intent); service advertisements are used to describe the services that a system provides, while service lookup requests document the services that are required. It is the role of the discovery enabler to capture this information from the legacy network protocols in use and to create an initial picture of the network systems wishing to connect with one another.

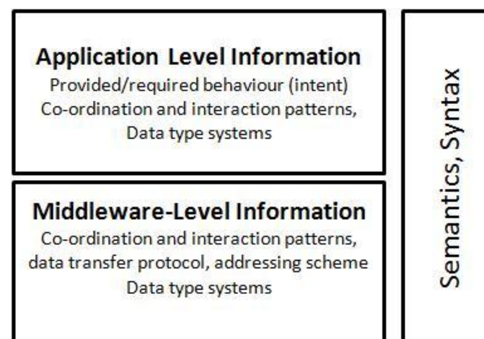


Figure 8.2: Networked System Model

The outputs of this enabler are models of networked system as shown in Figure 8.2. It is important to note that only a subset of this description is made available by the networked system; the learning enabler utilises an active learning algorithm to learn the co-ordination and interaction patterns of the application [60]. Much of the information about the middleware level is not explicit in the discovery process, but pointers within the discovery descriptions (e.g. this is a SOAP service) can be used to build the model from pre-defined, constant middleware models (e.g. a model of the SOAP protocol). The model builds upon discovery protocol descriptions that convey both syntactic information and semantic information about the externalized networked system. This semantic

information is necessary in open environments, where semantics cannot be assumed to be inherently carried in a commonly agreed syntax. Typically, ontologies are used in open environments for providing a common vocabulary on which semantic descriptions of networked systems can be based.

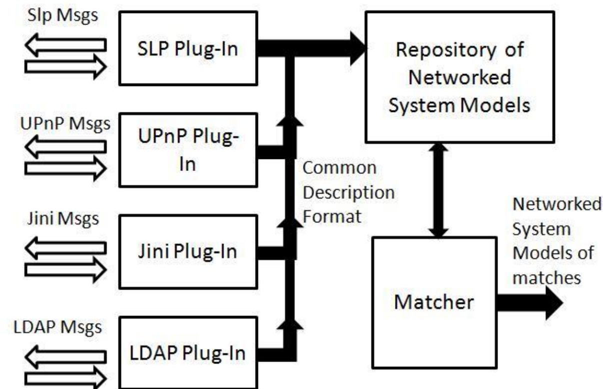


Figure 8.3: The Discovery Enabler

The architecture of the discovery enabler is illustrated in Figure 8.3. This software framework is deployed upon a third party node within the network and consists of three core elements:

- Discovery protocol plug-ins. Discovery protocols e.g. SLP, UPnP, LDAP, Jini, etc. are heterogeneous in terms of their behaviour and message format; further they differ in the data representation used to describe services. To resolve this, individual plug-ins for each protocol receive and send messages in the legacy format; the plug-in also translates the advertisements and requests into a common description format used by the CONNECT networked system model.

- The Model repository stores networked system models of all CONNECT ready systems (this is a system which advertises its intent and whose behaviour is learnable). These remains alive for the lifetime of the request-for a system advertising its services this will normally match the length of its lease as presented by the legacy protocol and, for a system's request, this is the length of time before the legacy protocol lookup request times out.

- The Functional Matcher actively matches potential requests with advertisements i.e. matching the required and provided interface types of a network system. Simple semantic matchers can be plugged into to match descriptions of the same type, or richer semantic matchers can be employed.

Synthesis of Connectors

We have already extensively describes the process to synthesize the connectors model in the previous chapters. Within the CONNECT context, the Connector Synthesis is a two-step process that encompasses the construction of a mediation LTS and its interpretation at runtime. The needed mediation LTS defines the behaviour that will let the networked systems synchronize and interact. It results from the analysis of both the networked systems behaviours and the ontology, and specifies all the needed message translations from one side to the other.
The mediation LTS resolves the application-level and data-level interoperability. The resulting mediation LTS remains an abstract specification that does not include enough middleware-level information to be directly executed.

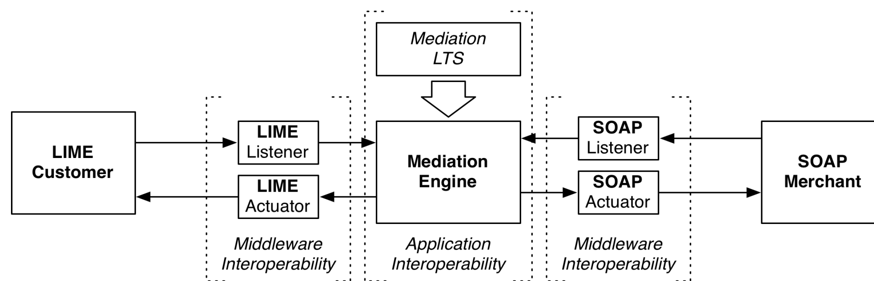


Figure 8.4: A software Connector

Instead, as shown on Figure 8.4, the mediation LTS is seen as an orchestration of middleware invocations and is dynamically interpreted by an engine, which receives, translates and forwards messages from the two sides. In our example, when the mediation engine is notified of a getInfo tuple was released by the client, it triggers the emission of three SOAP requests and triggers the generation of one Lime tuple containing the requested information. As shown in Figure 8.4, the missing middleware-level knowledge is hard-coded into reusable plug-ins denoted as Listener and Actuator. According to a given middleware protocol, a listener receives data packets and outputs application messages whereas an actuator composes network messages. In our marketplace example, the proper invocation of the Lime infrastructure and the emission and reception of SOAP messages are handled by those ad-hoc listeners and actuators. The use of such plug-ins finally ensures the middleware-level interoperability. In addition, when a new middleware is released, such plug-ins can be separately generated from the networked system models. By contrast with code-generation, the choice of interpretation eases the monitoring and dependability verification of runtime Connectors. Although the Connect framework also addresses these two issues, they are not presented here for the sake of conciseness.

8.1.4 CONCLUSION

We have shown that in spite of the major research and industrial efforts to solve the problem of interoperability, current solutions demonstrably fail to meet the needs of modern

distributed applications especially those that embrace dynamicity and high levels of heterogeneity. An important observation is that there is a significant disconnect between middleware solutions and semantic interoperability solutions, which in turn severely hampers progress in this area. We have introduced the CONNECT architecture as a fundamentally different way to address the interoperability problem; this intrinsically supports middleware and application level interoperability and embraces learning and synthesis. The initial experiment with the architecture provides early evidence of the validity of the proposed approach and we believe that as the architecture matures it will provide further novel and rich contributions to the field of interoperability. Future work will continue to explore a broader range of issues in the heterogeneity space. Much of this will focus on the important requirements that have not been yet investigated and their integration into the Connect software architecture. These include for example: i) non-functional properties. That is creating connectors that conform to the non-functional requirements of both interacting parties in the same way they meet the functional requirements currently; iii) Dependability. Ensuring that the deployed connectors are dependable, trustworthy and secure; this is especially important given the nature of the pervasive computing environments where these solutions will be deployed.

CHAPTER 9

DISCUSSION AND FUTURE WORK

Automated and on-the-fly interoperability is a key requirement for heterogeneous protocols within ubiquitous computing environments where networked systems *meet dynamically* and need to *interoperate without a priori knowledge* of each other.
Although numerous efforts has been done in many different research areas, such kind of interoperability is still an open challenge.
The research question on which this thesis focused is: "given compatible protocols, is it possible to automatically synthesize a mediator which enables them to communicate (solving their mismatches)?".
To answer to this question, we proposed techniques to automatically reason about and compose the behaviour of networked systems that aim at fulfilling some intent by connecting to other systems.

The reasoning serves to find a way to achieve communication -if it is possible- and to build the related mediation solution. Our work put the emphasis on "the elicitation of a way to achieve communication" while it can gain from more practical treatments of similar problems in the literature like converters, or adaptors or coordinators synthesis.
In particular, our work amounts to the following contributions:

1 Design of `AMAzING`, a comprehensive mediator synthesis process made by three phases: protocol abstraction, matching and mapping.

2 A set of mediator patterns which represent the building blocks to tackle in a systematic way the protocol mediation. This led us to devise a complete characterization of the protocol mismatches that we are able to solve by our connector synthesis process and to define significant mediator patterns as solution to the classified problems.

3 Formalization of `MediatorS`, a theory of emerging mediating connectors which includes related automated model-based techniques and tools to support the devised `AMAzING` synthesis process. The theory rigorously characterizes: (i) application layer protocols, (ii) their abstraction, (iii) the conditions under which two protocols are functionally matching, (iv) the notion of interoperability between protocols based on the definition of the functional matching relationship, and (v) the mapping, i.e., the synthesis of the mediator behaviour needed to achieve protocol interoperability under functional matching. We also discussed the correctness of the synthesized mediator.

4 An extension of the MediatorS theory for dealing also with middleware layer protocols and data, in addition to application layer protocols. Accordingly, we provided: (i) new models for rigorously characterizing networked systems (ii) new abstractions, (iii) new matching and (iv) mapping functions to reason upon networked systems compatibility, and a (iv) new mediator protocol definition.

Further, we positioned this thesis with respect to surveyed related works. MediatorS and AMAzING, among other case studies, have been applied in this thesis to Flickr and Picasa services and they have also been adopted by the European Research Project CONNECT. Moreover, we did a first effort towards taking into account also non-functional properties while building mediators which is described in [18] and summarized in the following.
To build an interoperability solution between the networked systems, two aspects have to be considered of the connected system under-construction: *functional interoperability* and *non-functional interoperability*. The first one solely refers to functional properties and aims at allowing the networked systems to communicate. Instead, non-functional interoperability refers to the assessment and achievement of the non-functional characteristics which qualify the communication (*how* it should be provided).
We proposed a combined interoperability approach made by the integration of our automated technique for the synthesis of mediators with a monitoring mechanism. The mediators provide functional interoperability and the monitors make it possible to assess the non-functional characteristics of the connected system at runtime that cannot be assessed statically at synthesis time. The combined approach then addresses functional interoperability pursued by-construction at synthesis time (i.e., a-priori), and non-functional interoperability, that is compliance to non-functional constraints, continuously assessed at execution time (a-posteriori), by passive monitoring.

In the following we conclude with a discussion that outlines *future work perspectives*. The mediator patterns described in Chapter 4 are twofold: (a) they are a set of design building blocks to tackle in a systematic way the protocol mediation problem and (b) they rigorously characterize the kind of interoperability mismatches we deal with. Patterns can hence serve as base for addressing compositional connector synthesis that has not been addressed so far. Then, as future works we intend to:

- refine the design of the compositional approach based on patterns, that we sketched in Section 4.2, by exploiting the algebra of connectors presented in [6].
- provide, in the direction of automated code generation: (i) the "concrete" Basic Mediator Patterns, i.e., the skeleton code corresponding to the "abstract" patterns presented in this chapter, (ii) the implementation of the pattern-based approach, i.e., the actual code for the component's behaviour decomposition and composition and the mediating connector behaviour building.

The MediatorS *theory* proposed in Chapter 5, and extended in Chapter 6, (1) clearly defines the interoperability problem, (2) shows the feasibility of the automated reasoning

about protocols, i.e., functional matching, and (3) shows the feasibility of the automated synthesis of abstract mediators under certain assumptions. In the future we aim to:

- implement the theory algorithms being able to automatize the mediator generation. In this direction we are currently working to on-the-fly reasoning about interoperability using ontology-based model checking [16];
- study run-time techniques towards efficient synthesis;
- scale the synthesis process. The current theory is described considering only two protocols but extending it to an arbitrary number n of protocols seems not to be be problematic. This aspect has to be still investigated. However, the protocol abstraction step developed within the devised AMAzING process represents a first attempt in this direction by reducing the size of the behavioural models of the NSs to be connected;
- extend the validation of the theory on other real world applications. It would possibly help in tuning the theory, if needed, and in refining the borders among which the theory works;
- translate the synthesized connector model into an executable artefact that can be deployed and run in the network for actual enactment of the connectors, as studied in the CONNECT project. This also requires devising the runtime architecture of connectors (preliminary achievements have been presented in Chapter 8) by investigating the issue of generation of code versus interpretation of the connector model;
- ensure dependability. Preliminary results towards this aim have been summarized above and are described in our paper [18]. Our combined interoperability approach is a first step in the direction of a complete interoperability solution, i.e., including both functional and non-functional interoperability. As future work, we plan to investigate the following aspects. Next to dynamic monitoring, we aim to take into account non-functional interoperability *during* the AMAzING process. Indeed we would include also the modeling of non-functional aspects, together with their respective matching and mapping reasoning.
  With respect to the current combined approach we also need to: propose a language to express non-functional constraints and properties; provide reaction policies or reaction policy patterns that can be undertaken when something wrong is detected by the monitoring. Examples are: to use predictive approaches that try to prevent the wrong behaviours; to adapt the CONNECT architectural infrastructure, if possible, for improving the provided connection; eventually, to notify the Networked Systems about the unexpected behaviour, and let them directly handle the problem.
- relax some assumptions, towards a dynamic environment, and manage the consequent uncertainty. For example, (i) instead of considering the abstraction ontology mapping given, we aim to infer it through an iterative interaction with the ontology.

Further, we aim at integrating with complementary works about: (ii) learning techniques to dynamically learn the protocols (instead of assuming them given) that are run in the environment; (iii) universal discovery techniques to dynamically perform the matching and mapping of protocols thus allowing on-the-fly interoperability; (iv) data-level interoperability (instead of assuming the ontology given) to elicit the data mappings. This may rise the problem of dealing with partial or erroneous specification.

REFERENCES

[1] Artix enterprise service bus software. http://web.progress.com/en/sonic/artix-esb.html, 2010.

[2] IBM Software WebSphere. http://www-01.ibm.com/software/websphere/.

[3] ABERER, K., CATARCI, T., CUDRÉ-MAUROUX, P., DILLON, T. S., GRIMM, S., HACID, M.-S., ILLARRAMENDI, A., JARRAR, M., KASHYAP, V., MECELLA, M., MENA, E., NEUHOLD, E. J., OUKSEL, A. M., RISSE, T., SCANNAPIECO, M., SALTOR, F., SANTIS, L. D., SPACCAPIETRA, S., STAAB, S., STUDER, R., AND TROYER, O. D. Emergent semantics systems. In *ICSNW* (2004), pp. 14–43.

[4] ABERER, K., CUDRÉ-MAUROUX, P., M.OUKSEL, A., CATARCI, T., HACID, M.-S., ILLARRAMENDI, A., KASHYAP, V., MECELLA, M., MENA, E., NEUHOLD, E. J., TROYER, O. D., RISSE, T., SCANNAPIECO, M., SALTOR, F., SANTIS, L. D., SPACCAPIETRA, S., STAAB, S., AND STUDER, R. Emergent semantics principles and issues. In *9th International Conference on Database Systems for Advances Applications* (2004), pp. 25–38.

[5] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[6] AUTILI, M., CHILTON, C., INVERARDI, P., KWIATKOWSKA, M. Z., AND TIVOLI, M. Towards a connector algebra. In *ISoLA (2)* (2010), pp. 278–292.

[7] AUTILI, M., INVERARDI, P., NAVARRA, A., AND TIVOLI, M. SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems. In *International Conference on Software Engineering (ICSE'07)* (Los Alamitos, CA, USA, 2007), IEEE Computer Society, pp. 784–787.

[8] AUTILI, M., MOSTARDA, L., NAVARRA, A., AND TIVOLI, M. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software 81*, 12 (2008), 2210–2236.

[9] AVGERIOU, P., AND ZDUN, U. Architectural Patterns Revisited - a Pattern Language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)* (Irsee, Germany, July 2005).

[10] BAADER, F., CALVANESE, D., MCGUINNESS, D. L., NARDI, D., AND PATEL-SCHNEIDER, P. F. *The Description Logic Handbook*. Cambridge University Press, 2003.

[11] BALEK, D. *Connectors in Software Architectures*. PhD thesis, Charles University, Prague - Czech Republic, May 2002.

[12] BARBOSA, M. A., AND BARBOSA, L. S. Specifying software connectors. In *ICTAC* (2004), pp. 52–67.

[13] BEN MOKHTAR, S., PREUVENEERS, D., GEORGANTAS, N., ISSARNY, V., AND BERBERS, Y. EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. *Journal of Systems and Software 81*, 5 (2008), 785–808.

[14] BENATALLAH, B., CASATI, F., GRIGORI, D., NEZHAD, H. R. M., AND TOUMANI, F. Developing adapters for web services integration. In *proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE), Porto, Portugal* (2005), Springer Verlag, pp. 415–429.

[15] BENNACEUR, A., BLAIR, G. S., CHAUVEL, F., GEORGANTAS, N., GRACE, P., HOWAR, F., INVERARDI, P., ISSARNY, V., PAOLUCCI, M., PATHAK, A., SPALAZZESE, R., STEFFEN, B., AND SOUVILLE, B. Towards an architecture for runtime interoperability. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2010), Springer.

[16] BENNACEUR, A., ISSARNY, V., AND SPALAZZESE, R. On-the-fly reasoning about interoperability using ontology-based model checking. technical report, inria rocquencourt, paris., Jan. 2011.

[17] BENNACEUR, A., SPALAZZESE, R., INVERARDI, P., ISSARNY, V., GEORGANTAS, N., AND SAADI, R. Model-based mediators for dynamic-adaptive connectors. Tech. rep., INRIA Paris-Rocquencourt, France. January 2011.

[18] BERTOLINO, A., INVERARDI, P., ISSARNY, V., SABETTA, A., AND SPALAZZESE, R. On-the-fly interoperability through automated mediator synthesis and monitoring. In *ISoLA 2010, Part II, LNCS 6416,* (2010), Springer, Heidelberg, pp. 251–262.

[19] BERTOLINO, A., INVERARDI, P., PELLICCIONE, P., AND TIVOLI, M. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2009), ACM, pp. 141–150.

[20] BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C., AND OR-CHARD, D. Web services architecture. in `http://www.w3.org/TR/sawsdl/`. w3c, february 2004.

[21] BOTH, A., AND ZIMMERMANN, W. Automatic protocol conformance checking of recursive and parallel bpel systems. In *ECOWS* (2008), pp. 81–91.

[22] BOTH, A., ZIMMERMANN, W., AND FRANKE, R. Model checking of component protocol conformance - optimizations by reducing false negatives. *Electr. Notes Theor. Comput. Sci. 263* (2010), 67–94.

[23] BRACCIALI, A., BROGI, A., AND CANAL, C. A formal approach to component adaptation. *Journal of Systems and Software 74*, 1 (2005), 45–54.

[24] BROGI, A., CANAL, C., AND PIMENTEL, E. Behavioural types and component adaptation. In *Algebraic Methodology and Software Technology: 10th International Conference, AMAST 2004, volume 3116 / 2004 of LNCS* (2004), Springer-Verlag GmbH, pp. 42–56.

[25] BROMBERG, Y.-D., AND ISSARNY, V. INDISS: Interoperable Discovery System for Networked Services. In *Middleware* (2005), pp. 164–183.

[26] BROMBERG, Y.-D., AND ISSARNY, V. Formalizing middleware interoperability: From design time to runtime solutions. Tech. rep., Rocquencourt, France, 2008.

[27] BRUNI, R., LANESE, I., AND MONTANARI, U. A basic algebra of stateless connectors. *Theor. Comput. Sci. 366*, 1 (2006), 98–120.

[28] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[29] CALVERT, K. L., AND LAM, S. S. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communications 8*, 1 (1990), 127–142.

[30] CANAL, C., POIZAT, P., AND SALAÜN, G. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Software Eng. 34*, 4 (2008), 546–563.

[31] CAVALLARO, L., NITTO, E. D., AND PRADELLA, M. An automatic approach to enable replacement of conversational services. In *ICSOC/ServiceWave* (2009).

[32] CHAPPELL., D. *Enterprise Service Bus*. O'Reilly, 2004.

[33] CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. Web services description language (wsdl) 1.1. in `http://www.w3.org/TR/wsdl`, march 2001.

[34] CIMPIAN, E., AND MOCAN, A. Wsmx process mediation based on choreographies. In *Business Process Management Workshops* (2005), C. Bussler and A. Haller, Eds., vol. 3812, pp. 130–143.

[35] CONNECT. CONNECT Annex I: Description of Work. FET IP CONNECT EU project, FP7 grant agreement number 231167, `http://connect-forever.eu/`.

[36] CONNECT. CONNECT Deliverable D1.1: Initial Connect Architecture. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[37] CONNECT. CONNECT Deliverable D1.2: Intermediate Connect Architecture. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[38] CONNECT. CONNECT Deliverable D3.1: Modeling of application- and middleware-layer interaction protocols. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[39] CONNECT. CONNECT Deliverable D3.2: Reasoning about and harmonizing the interaction behavior of networked systems at application- and middleware- layer. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[40] DENARO, G., PEZZE', M., AND TOSI, D. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of ESEC/FSE 2009* (2009), ACM Press.

[41] DI GIANDOMENICO, F., KWIATKOWSKA, M., MARTINUCCI, M., MASCI, P., AND QU, H. Dependability analysis and verification for connected systems. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2010), Springer.

[42] DRUMMOND, N., RECTOR, A. L., STEVENS, R., MOULTON, G., HORRIDGE, M., WANG, H., AND SEIDENBERG, J. Putting OWL in order: Patterns for sequences in OWL. In *OWLED* (2006).

[43] DRY-RUN, C. The popcorn scenario. https://www-roc.inria.fr/connect/connect-dry-run/.

[44] DUFTLER, M. J., MUKHI, N. K., SLOMINSKI, A., SLOMINSKI, E., AND WEERAWARANA, S. Web Services Invocation Framework (WSIF). In *OOPSLA Workshop on Object Oriented Web Services* (2001).

[45] DUMAS, M., SPORK, M., AND WANG, K. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Business Process Management* (2006), pp. 65–80.

[46] EUZENAT, J., AND SHVAIKO, P. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[47] FARRELL, J., AND LAUSEN, H. Semantic annotations for wsdl and xml schema. http://www.w3.org/TR/sawsdl/, august 2007.

[48] FIADEIRO, J. L., LOPES, A., AND WERMELINGER, M. Theory and practice of software architectures. Tutorial at the 16th IEEE Conference on Automated Software Engineering, San Diego, CA, USA, Nov. 26-29, 2001.

[49] FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[50] FLICKR. `http://www.flickr.com/`.

[51] FLICKR API. `http://www.flickr.com/services/api/`.

[52] FLORES-CORTÉS, C. A., BLAIR, G. S., AND GRACE, P. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *IEEE Distributed Systems Online 8* (July 2007).

[53] FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In *ICSE* (2006), pp. 771–774.

[54] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[55] GANG, H., HONG, M., QIAN-XIANG, W., AND FU-QING, Y. A systematic approach to composing heterogeneous components. chinese journal of electronics, 12(4):499-505, 2003.

[56] GRACE, P., BLAIR, G. S., AND SAMUEL, S. ReMMoC: A reflective middleware to support mobile client interoperability. In *CoopIS/DOA/ODBASE* (2003), pp. 1170–1187.

[57] GROUP, O. M. The common object request broker: Architecture and specification version 2.0, 1995.

[58] GUDGIN, M., HADLEY, M., MENDELSOHN, N., MOREAU, J., NIELSEN, H. F., KARMARKAR, A., AND LAFON., Y. Soap version 1.2 part 1: Messaging framework. in `http://www.w3.org/TR/soap12-part1`, april 2001.

[59] HIRSCH, D., UCHITEL, S., AND YANKELEVICH, D. Towards a periodic table of connectors. In *Coordination Models and Languages (COORDINATION '99)* (1999), p. 418.

[60] HOWAR, F., JONSSON, B., MERTEN, M., STEFFEN, B., AND CASSEL, S. On handling data in automata learning: Considerations from the connect perspective. In *proceedings of ISOLA 2010* (2010).

[61] INTRIGILA, B., INVERARDI, P., AND ZILLI, M. V. A comprehensive setting for matching and unification over iterative terms. *Fundam. Inform. 39*, 3 (1999), 273–304.

[62] INVERARDI, P., ISSARNY, V., AND SPALAZZESE, R. A theory of mediators for eternal connectors. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Part II.* (2010), vol. 6416, Springer, Heidelberg, pp. 236–250.

[63] INVERARDI, P., AND NESI, M. Deciding observational congruence of finite-state ccs expressions by rewriting. *Theor. Comput. Sci. 139*, 1-2 (1995), 315–354.

[64] ISSARNY, V., STEFFEN, B., JONSSON, B., BLAIR, G., GRACE, P., KWIATKOWSKA, M., CALINESCU, R., INVERARDI, P., TIVOLI, M., BERTOLINO, A., AND SABETTA, A. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *14th IEEE International Conference on Engineering of Complex Computer Systems* (Postdam Germany, 2009).

[65] JIANG, F., FAN, Y., AND ZHANG, X. Rule-based automatic generation of mediator patterns for service composition mismatches. In *Proceedings of the 2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 3–8.

[66] KALFOGLOU, Y., AND SCHORLEMMER, M. Ontology mapping: the state of the art. *Knowl. Eng. Rev. 18*, 1 (January 2003), 1–31.

[67] KALFOGLOU, Y., AND SCHORLEMMER, M. Ontology mapping: The state of the art. In *Semantic Interoperability and Integration* (Dagstuhl, Germany, 2005), Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, Eds., no. 04391 in Dagstuhl Seminar Proceedings, (IBFI), Schloss Dagstuhl, Germany.

[68] KANADA, Y. Emergent computation. `http://www.kanadas.com/CCM/hyper/emergent-computation.html`, 1995.

[69] KELL, S. Rethinking software connectors. In *SYANCO '07: International workshop on Synthesis and analysis of component connectors* (New York, NY, USA, 2007), ACM, pp. 1–12.

[70] KELLER, R. M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (1976), 371–384.

[71] KUMAR, R., NELVAGAL, S., AND MARCUS, S. I. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems 7*, 3 (1997).

[72] LAM, S. S. Correction to "protocol conversion". *IEEE Trans. Software Eng. 14*, 9 (1988), 1376.

[73] LIMAM, N., ZIEMBICKI, J., AHMED, R., IRAQI, Y., LI, D. T., BOUTABA, R., AND CUERVO, F. Osda: Open service discovery architecture for efficient cross-domain service provisioning. *Comput. Commun. 30* (February 2007), 546–563.

[74] LISKOV, B., AND WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[75] LOPES, A., WERMELINGER, M., AND FIADEIRO, J. L. Higher-order architectural connectors. *ACM Trans. Softw. Eng. Methodol. 12*, 1 (2003), 64–104.

[76] MAGEE, J., AND KRAMER, J. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.

[77] MARTIN, D., BURSTEIN, M., MCDERMOTT, D., MCILRAITH, S., PAOLUCCI, M., SYCARA, K., MCGUINNESS, D. L., SIRIN, E., AND SRINIVASAN, N. Bringing semantics to web services with owl-s. *World Wide Web 10* (September 2007), 243–277.

[78] MEHTA, N. R., MEDVIDOVIC, N., AND PHADKE, S. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ACM Press, pp. 178–187.

[79] MILNER, R. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[80] MOTAHARI NEZHAD, H. R., BENATALLAH, B., MARTENS, A., CURBERA, F., AND CASATI, F. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web* (New York, NY, USA, 2007), ACM, pp. 993–1002.

[81] MOTAHARI NEZHAD, H. R., XU, G. Y., AND BENATALLAH, B. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 731–740.

[82] NAKAZAWA, J., TOKUDA, H., EDWARDS, W. K., AND RAMACHANDRAN, U. A bridging framework for universal interoperability in pervasive systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 3.

[83] NAKAZAWA, J., TOKUDA, H., EDWARDS, W. K., AND RAMACHANDRAN, U. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 2006), ICDCS '06, IEEE Computer Society, pp. 3–13.

[84] NEJATI, S., SABETZADEH, M., CHECHIK, M., EASTERBROOK, S., AND ZAVE, P. Matching and merging of statecharts specifications. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 54–64.

[85] NOY, N. F. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec. 33* (December 2004), 65–70.

[86] OKUMURA, K. A formal protocol conversion method. In *SIGCOMM* (1986), pp. 30–37.

[87] OLSON, R., AND SEQUEIRA, R. Emergent computation and the modeling and management of ecological systems. *Comput.Electron. Agric. 12*, 3 (1995), 183–209.

[88] OLSON, R., AND SEQUEIRA, R. An emergent computational approach to the study of ecosystem dynamics. *Ecological Model 79*, 1-3 (1995), 95–120.

[89] (OMG). COM/CORBA interworking specification Part A & B, 1997.

[90] P. CUDRÉ-MAUROUX AND K. ABERER (EDS), ABDELMOTY, A. I., CATARCI, T., DAMIANI, E., ILLARAMENDI, A., JARRAR, M., MEERSMAN, R., NEUHOLD, E. J., PARENT, C., SATTLER, K.-U., SCANNAPIECO, M., SPACCAPIETRA, S., SPYNS, P., AND TRÉ, G. D. Viewpoints on emergent semantics. *Journal on Data Semantics IV* (2006), 1–27.

[91] PAOLUCCI, M., KAWAMURA, T., PAYNE, T. R., AND SYCARA, K. P. Semantic matching of web services capabilities. In *ISWC* (2002).

[92] PERRY, D. E., AND WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes 17* (October 1992), 40–52.

[93] PICASA. http://picasaweb.google.com/.

[94] PICASA API. http://code.google.com/intl/it-IT/apis/picasaweb/docs/2.0/developers_guide_java.html.

[95] PICASA EXAMPLE. http://code.google.com/p/gdata-java-client/downloads/list.

[96] PLOTKIN, G. D. A note on inductive generalization. *Machine Intelligence 5* (1970), 153–163.

[97] PONNEKANTI, S., AND FOX, A. Interoperability among independently evolving Web services. In *Proc. ACM/IFIP/USENIX Middleware Conference* (2004), pp. 331–351.

[98] REYNOLDS, J. Transformational systems and the algebraic structure of atomic formulas machine intelligence, edinburgh university press, usa, vol. 5, pp. 135-151, 1970.

[99] ROMAN, D., KELLER, U., LAUSEN, H., DE BRUIJN, J., LARA, R., STOLLBERG, M., POLLERES, A., FEIER, C., BUSSLER, C., AND FENSEL, D. Web service modeling ontology. *Appl. Ontol. 1* (January 2005), 77–106.

[100] ROMAN, M., KON, F., AND CAMPBELL, R. H. Reflective middleware: From your desk to your hand. Tech. rep., Champaign, IL, USA, 2000.

[101] ROSCOE, A. W., HOARE, C. A. R., AND BIRD, R. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[102] RUSKIN, H. J., AND WALSHE, R. Emergent computing - introduction to the special theme. http://www.ercim.org/publication/Ercim_News/enw64/intro-st.html, Jan, 2006.

[103] SATYANARAYANAN, M. Pervasive computing: vision and challenges. *IEEE Personal Communications 8*, 4 (2001), 10–17.

[104] SCHREINER, D., AND GÖSCHKA, K. M. Building component based software connectors for communication middleware in distributed embedded systems. In *Proceedings of the '3rd ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA 2007), Las Vegas, Nevada, USA* (2007).

[105] SHAW, M. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Selected papers from the Workshop on Studies of Software Design* (London, UK, 1996), ICSE '93, Springer-Verlag, pp. 17–32.

[106] SHAW, M., AND GARLAN, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[107] SPALAZZESE, R. Towards mediator connectors for application level interoperability. In *ESEC/FSE Doctoral Symposium '09: Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium* (New York, NY, USA, 2009), ACM, pp. 35–36.

[108] SPALAZZESE, R., AND INVERARDI, P. Components interoperability through mediating connector pattern. In *WCSI 2010, arXiv:1010.2337; EPTCS 37, 2010, pp. 27-41*.

[109] SPALAZZESE, R., AND INVERARDI, P. Mediating connector patterns for components interoperability. In *ECSA* (2010), pp. 335–343.

[110] SPALAZZESE, R., INVERARDI, P., AND ISSARNY, V. Towards a formalization of mediating connectors for on the fly interoperability. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009)* (2009), pp. 345–348.

[111] SPITZNAGEL, B. *Compositional Transformation of Software Connectors*. PhD thesis, Carnegie Mellon University, May 2004.

[112] SPITZNAGEL, B., AND GARLAN, D. A compositional formalization of connector wrappers. In *ICSE* (2003), pp. 374–384.

[113] STOLLBERG, M., CIMPIAN, E., MOCAN, A., AND FENSEL, D. A semantic web mediation architecture. In *In Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006* (2006), Springer.

[114] STUDER, R., BENJAMINS, V. R., AND FENSEL, D. Knowledge engineering: Principles and methods. *Data Knowl. Eng. 25*, 1-2 (1998), 161–197.

[115] SUH, N., BASS, B., CHAN, E., AND TOLLER, N. Emergent computation and modelling: Complex organization and bifurcation within environmental bounds (cobweb). *Journal of Environmental Informatics 1*, 2 (2003. ISEIS), 1–11.

[116] TAYLOR, R. N., MEDVIDOVIC, N., AND DASHOFY, E. M. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[117] TIVOLI, M., AND INVERARDI, P. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program. 71* (May 2008), 181–212.

[118] VACULÍN, R., NERUDA, R., AND SYCARA, K. P. An agent for asymmetric process mediation in open environments. In *SOCASE* (2008), R. Kowalczyk, M. N. Huhns, M. Klusch, Z. Maamar, and Q. B. Vo, Eds., vol. 5006 of *Lecture Notes in Computer Science*, Springer, pp. 104–117.

[119] VACULÍN, R., AND SYCARA, K. Towards automatic mediation of OWL-S process models. *Web Services, IEEE International Conference on 0* (2007), 1032–1039.

[120] WATT, S. M. Algebraic generalization. *SIGSAM Bull. 39*, 3 (2005), 93–94.

[121] WEISER, M. The computer for the 21st century. *Scientific American* (Sep. 1991).

[122] WEISER, M. Hot Topics: Ubiquitous Computing. *IEEE Computer* (oct 1993).

[123] WEISER, M. Ubiquitous computing. http://sandbox.xerox.com/ubicomp/, 1996.

[124] WERMELINGER, M., AND FIADEIRO, J. L. Connectors for mobile programs. *IEEE Trans. Softw. Eng. 24*, 5 (1998), 331–341.

[125] WIEDERHOLD, G. Mediators in the architecture of future information systems. *IEEE Computer 25* (1992), 38–49.

[126] WIEDERHOLD, G., AND GENESERETH, M. The conceptual basis for mediation services. *IEEE Expert: Intelligent Systems and Their Applications 12*, 5 (1997), 38–47.

[127] WILLIAMS, S. K., BATTLE, S. A., AND CUADRADO, J. E. Protocol mediation for adaptation in semantic web services. In *ESWC* (2006), pp. 635–649.

[128] WOOLLARD, D., AND MEDVIDOVIC, N. High performance software architectures: A connector-oriented approach. In *Proceedings of the Institute for Software Research Graduate Research Symposium, Irvine, California* (June 2006).

[129] XITONG, L., YUSHUN, F., JIAN, W., LI, W., AND FENG, J. A pattern-based approach to development of service mediators for protocol mediation. In *proceedings of WICSA '08* (2008), IEEE Computer Society, pp. 137–146.

[130] YELLIN, D. M., AND STROM, R. E. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst. 19*, 2 (1997), 292–333.

[131] ZHU, F., MUTKA, M. W., AND NI, L. M. Service discovery in pervasive computing environments. *IEEE Pervasive Computing 4*, 4 (2005), 81–90.