

Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware- Layer

Amel Bennaceur, Franck Chauvel, Paola Inverardi, Valerie Issarny, Ilaria Matteucci, Fabio Martinelli, Romina Spalazzese, Massimo Tivoli

▶ **To cite this version:**

Amel Bennaceur, Franck Chauvel, Paola Inverardi, Valerie Issarny, Ilaria Matteucci, et al.. Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware- Layer. [Research Report] 2011. inria-00584917

HAL Id: inria-00584917
https://inria.hal.science/inria-00584917

Submitted on 11 Apr 2011

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.

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D3.2

Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware-Layer



SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

INRIA docomo DOCOMO Euro-Labs LANCASTER UNIVERSITY THALES Universita' dell'Aquila technische universität dortmund UNIVERSITY OF OXFORD UPPSALA UNIVERSITET

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	Report, Prototype

Deliverable Number	:	D3.2
Title of Deliverable	:	Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware-Layer
Nature of Deliverable	:	Report, Prototype
Dissemination Level	:	Public
Internal Version Number	:	0.1
Contractual Delivery Date	:	31 January 2011
Actual Delivery Date	:	21 February 2011
Contributing WPs	:	WP3
Editor(s)	:	Massimo Tivoli (UNIVAQ)
Author(s)	:	Amel Bennaceur (INRIA), Franck Chauvel (PKU), Paola Inverardi (UNIVAQ), Valérie Issarny (INRIA), Ilaria Matteucci (CNR), Fabio Martinelli (CNR), Romina Spalazzese (UNIVAQ), Massimo Tivoli (UNIVAQ)
Reviewer(s)	:	Nikolaos Georgantas (INRIA)

Abstract

The CONNECT *Integrated Project aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from 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 behavioral semantics of application- down to middleware-layer protocols run by the interacting parties.*

The role of work package WP3 is to devise automated and compositional approaches to connector synthesis, which can be performed at run-time. Given the respective interaction behavior of networked systems, we want to synthesize the behavior of the connector(s) needed for them to interact. These connectors serve as mediators *of the networked systems' interaction at both application and middleware layers.*

During the project's first year, the work of WP3 led us to achieve the following preliminary results: the formalization of matching and mapping relationships for application-layer interaction protocols; the definition of the corresponding mediator generation algorithm; the analysis of the interoperability problems, and related solutions, that can occur at middleware-layer; and a model-driven approach to the automated elicitation of application-layer protocols from software implementations. All these achievements have been reported in Deliverable D3.1: "Modeling of application- and middleware-layer interaction protocols".

In this deliverable, we go a step forward with respect to some of the previous achievements by delivering a unified process, and related artefacts, for the automated synthesis of mediators at both application and middleware layers, code-generation techniques to generate the actual code that implements a synthesized mediator, and a preliminary integration of QoS management in the synthesis process. During year 2, all the work has been validated through its application to several scenarios, in particular as part of WP1 and WP6. By selecting one of them as common scenario, in this deliverable, we also show the different methods/techniques at work on the scenario. All the steps of the devised synthesis process are described in detail and applied to the selected common scenario.

Keyword List

Connectors, Protocol Mediators, Protocol Specification, Protocol Synthesis, Application-Layer Interoperability, Middleware-Layer Interoperability, Security, Code Generation.

Table of Contents

LIST OF FIGURES 9

LIST OF TABLES 11

1 INTRODUCTION 13

1.1 The Role of Work Package WP3 14
1.2 Brief Summary of Achievements in Year 1 14
1.3 First Review Recommendations and Related Reactions for Year 2 15
1.4 Overview of the Unified CONNECTor Synthesis Process 16
1.5 Illustrative Example: the Photo Sharing Scenario 18
1.6 Challenges for Year 2 and Overview of the Related Achievements 20

2 MEDIATOR PATTERNS 23

2.1 Mediating Connector Architectural Pattern 25
2.2 Basic Mediator Patterns 27
2.3 Application of the Patterns to the Photo Sharing Scenario 33
2.4 Related Work 34
2.5 Conclusion 35

3 A THEORY OF MEDIATORS FOR ETERNAL CONNECTORS 37

3.1 Towards Emergent Mediators 38
3.2 A Formalization of Protocols 39
3.3 Abstract Protocol 41
3.4 Towards Automated Matching and Mediator Synthesis 44
3.5 Implementing the Theory 45
3.5.1 Abstraction Algorithms 46
3.5.2 Matching Algorithms 47
3.5.3 Mapping Algorithms 48
3.6 Related Work 49
3.7 Conclusion 50

4 ABSTRACT CONNECTOR SYNTHESIS 51

4.1 Modeling Networked Systems 52
4.1.1 Affordance 52
4.1.2 Interface Signature 53
4.1.3 Affordance Protocol 54
4.2 Ontology for Mediation 56
4.2.1 Middleware Ontology 57
4.2.2 Application-specific Ontology 59
4.3 Functional Matching of Networked Systems 60
4.3.1 Semantic Matching of Affordances 60
4.3.2 Interface Mapping 62
4.3.3 Behavioral Matching of Affordances 63
4.4 Mediator Synthesis 64
4.5 Related Work 64
4.6 Conclusion 66

5 FROM ABSTRACT TO CONCRETE MEDIATOR SYNTHESIS AND DEPLOYMENT 67
5.1 Overview 67
5.1.1 Run-Time Architecture of a CONNECTor 67
5.1.2 Interaction with the Interoperability Proxies 68
5.1.3 Code Generation vs. Model Interpretation 68
5.2 Modeling Run-Time Connectors 70
5.3 Generation of ad hoc Connectors 70
5.3.1 Existing Code Generation Strategies for LTS Models 71
5.3.2 Example: Using Nested Switch Statements 73
5.3.3 Compiling and Packaging CONNECTors 75
5.3.4 Deploying and Starting up CONNECTors 75
5.3.5 Prototype Tool Support 76
5.4 Towards Model Interpretation 76
5.4.1 CONNECTors vs. Service Orchestrations 77
5.4.2 From CONNECTors to BPEL Processes 78
5.4.3 Towards a Multi-Protocols BPEL Engine 80
5.5 Application to the Photo Sharing Scenario 81
5.6 Conclusion 84
6 DEALING WITH NON-FUNCTIONAL PROPERTIES 87
6.1 Compositional analysis: the partial model checking 87
6.2 Securing the Connector by Differential Controller 88
6.2.1 Application to the Photo-Sharing scenario 92
6.3 Conclusion and Future Work 92
7 CONCLUSION AND FUTURE WORKS 93
BIBLIOGRAPHY 95

List of Acronyms

Acronym	Meaning
AXIS	Apache EXtensible Interaction System
BPEL	Business Process Execution Language
BPMN	Business Process Modelling Notation
DSL	Domain Specific Language
IB	Infrastructure Based
ODE	Orchestration Director Engine
OSGi	Open Services Gateway Initiative
JAR	Java ARchive
JET	Java Emmitter Templates
LTS	Labeled Transition System
LIME	Linda in a Mobile Environment
NS	Networked System
P2P	Peer-to-Peer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
WSDL	Web Service Description Language
WS	Web Service

List of Figures

Figure 1.1: Data-flow in the CONNECT system 13

Figure 1.2: Elicitation of middleware-agnostic protocols 17

Figure 1.3: Abstracting protocol actions 17

Figure 1.4: Existence and subsequent synthesis of a CONNECTor 18

Figure 1.5: Concrete CONNECTor generation 19

Figure 1.6: Code generation 19

Figure 1.7: Overview of the Photo Sharing scenario 20

Figure 2.1: Peer-to-Peer-based (P2P) implementation 24

Figure 2.2: Infrastructure-based (IB) implementation 24

Figure 2.3: Entities involved in a mediated system 26

Figure 2.4: Basic interoperability mismatches 27

Figure 2.5: Basic solutions for the basic mismatches 28

Figure 2.6: Variants of the Basic Mediator Pattern (1) 29

Figure 2.7: Variants of the Basic Mediator Pattern (2) 30

Figure 2.8: Variants of the Basic Mediator Pattern (3) 31

Figure 2.9: Variants of the Basic Mediator Pattern (4) 32

Figure 2.10: Behavioral description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 2.2 a) and P2P Photo Sharing version 1 of Figure 2.1 v1)) 33

Figure 3.1: An overview of our approach 38

Figure 3.2: Ontology mapping and common language between Infrastructure-based Photo-Sharing Producer and the peer-to-peer Photo-Sharing version 1 (Figure 2.2 a) and Figure2.1 v1) respectively) 41

Figure 3.3: Abstract protocol building 42

Figure 3.4: Abstracted LTSs of the Photo Sharing protocols 43

Figure 4.1: Infrastructure- and peer-to-peer-based Photo Sharing 55

Figure 4.2: Middleware ontology 56

Figure 4.3: Middleware alignment 58

Figure 4.4: Shared-memory based Photo Sharing 58

Figure 4.5: Middleware-agnostic peer-to-peer Photo Sharing 59

Figure 4.6: Photo Sharing ontology 60

Figure 5.1: Run-time Architecture of CONNECTors 68

Figure 5.2: Synchronous communication pattern between the CONNECTor Core and the Interoperability Proxies 69

Figure 5.3: Asynchronous communication pattern between the CONNECTor Core and the Interoperability Proxies 69

Figure 5.4: Compilation of CONNECTor models 69

Figure 5.5: Runtime interpretation of CONNECTor models 69

Figure 5.6: Syntactic Domain of Run-time CONNECTors 71

Figure 5.7: Detailed design of a mediator implemented using the nested switch pattern 73

Figure 5.8: Compiling and Packaging CONNECTor source files into an OSGi bundle 76

Figure 5.9: Building the code generator from code templates using JET 77

Figure 5.10: Internal Architecture of the BPEL Apache ODE Engine 81

Figure 5.11: A possible extension of the Apache ODE Engine to support communication with various middeware technologies 81

Figure 5.12: Expected behavior of the LIME photo producer 81

Figure 5.13: Expected behavior of the SOAP-RPC server 81

Figure 5.14: LTS resulting from the synthesis and capturing the needed behavior to properly mediate between the LIME producer and the SOAP server 82

Figure 5.15: Equivalent Mealy machine, representing the mediation behavior 82

Figure 6.1: Partial evaluation function for parallel operator, where $t = (S, Act, \rightarrow)$ is the labelled transition system representing a process while $s \in S$ is the state in which we evaluate the formula [9]. 88

List of Tables

Table 5.1: Overview of the possible implementation patterns, with respect to the existing branching mechanisms 72

Table 6.1: Semantics definition of controller operators for enforcing safety properties. 90

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of NSs to respond to the evolution of functionalities provided to and required from 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 NSs communicate. The resulting emergent connectors (or CONNECTors) are effectively synthesized according to the behavioral semantics of application- down to middleware-layer protocols run by the interacting parties. The role of work package WP3 is to devise automated and compositional approaches to CONNECTor synthesis, which can be performed at run-time. Given the respective interaction behavior of NSs, we want to synthesize the behavior of the CONNECTor(s) needed for them to interact. These CONNECTors serve as *mediators* of the NS interaction at both application and middleware layers.

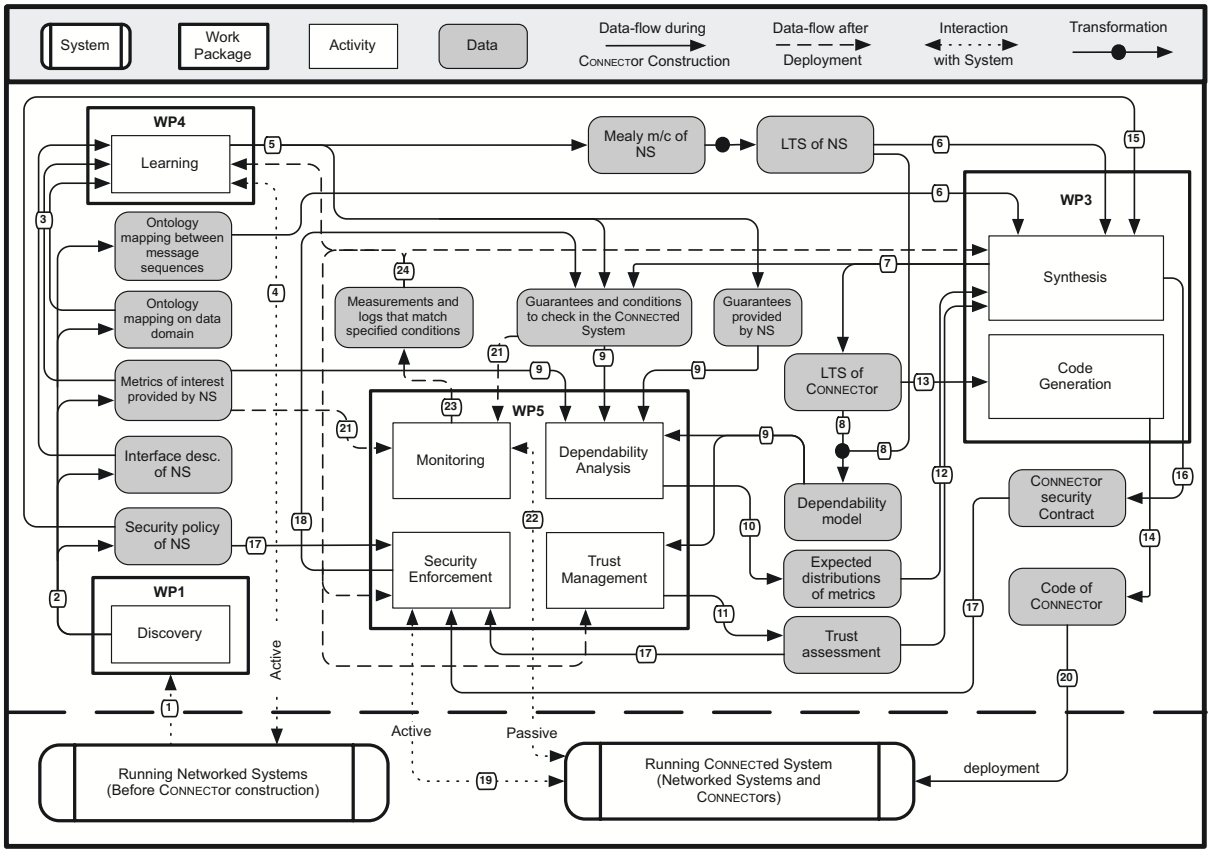


Figure 1.1: Data-flow in the CONNECT system

A high level view of the CONNECT system is described, in Section 5.1 of Deliverable D1.1 [2], as a system of various enablers that exchange information about the NS to be constructed, as depicted in Figure 1.1. In particular, the *Synthesis* enabler (represented by the box labeled "WP3: Synthesis") takes as input a high-level description (functional and non-functional) of the NSs to be CONNECTed and of the assumptions they make on the expected environment. This description employs different kinds of models, from behavioral models to ontologies and quantitative models expressing different aspects of an NS. These models are provided by the *Learning* and *Discovery* enablers and, to manage run-time evolution, they are also provided/updated by the *WP5* enablers. From these inputs, the *Synthesis* enabler automatically synthesizes a model of the CONNECTor that serves as input for the *Code Generation* enabler (represented by the box labeled "WP3: Code Generation") in order to automatically generate the actual code that

implements the synthesized CONNECTor.

A major challenge for WP3 in CONNECT is to develop a comprehensive, automated synthesis process taking into account both control and data aspects of a NS behavior, and also dealing with non-functional aspects, at both application and middleware layers. Furthermore, the synthesis process should be carried on at run-time, hence dealing also with CONNECTor evolution, scalability, and compositionality issues. Last, concerning code generation, the challenge is to prevent human intervention as much as possible by hopefully making the generation step completely automated.

1.1 The Role of Work Package WP3

Here, we simply recall that the role of WP3 is to [1]: *"devise automated and compositional approaches to connector synthesis, which can be performed at run-time. Given the respective interaction behavior of NSs, we want to synthesize the behavior of the wrapper(s) needed for them to interact. These wrappers have to serve as* mediators *of the networked applications' interaction at both the application- and middleware-layer"*. More specifically, WP3 has three main objectives that can be summarized as follows:

- **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.
- **Synthesis of middleware-layer protocols**. Our 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.
- **Model-driven synthesis tools**. In this subtask, we exploit model-to-model and model-to-code transformation techniques to automatically derive, at run-time, a connector's actual code from its synthesized model. This step should guarantee the *correctness-by-construction* of the connectors' implementations with respect to the functional and non-functional requirements of the networked applications that are made interoperable through the connectors.

1.2 Brief Summary of Achievements in Year 1

During the first year of the project, the work of WP3 led us to achieve two main results as summarized below. These achievements are described in detail in Deliverable D3.1 [5].

- A preliminary definition and **formalization of protocol matching and mapping relationships over application-layer protocols**. This contribution has been applied to two case studies: one concerned interoperability between two heterogeneous instant messaging systems, while the other concerned the application of the approach to the *Popcorn scenario* provided by WP1 and described in Deliverable D1.1 [2]. We recall that the defined relationships represent two essential operations for the dynamic synthesis of mediating connectors to enable eternal NSs. In fact, the matching relationship allows the rigorous characterization of the conditions that must hold in order for two heterogeneous protocols to be able to interoperate through a mediator. Thus, it allows one to state/check the existence of a mediator for two heterogeneous protocols. The mapping relationship introduces the formal specification of the algorithm that should be performed in order to automatically synthesize the required mediator.
- An analysis of the different dimensions of middleware-layer interoperability. This analysis led us to produce a **formalization of existing solutions to middleware-layer interoperability** and assess the one based on dynamic protocol synthesis as aimed by CONNECT through two examples taken from the *Popcorn scenario*. This led us to conclude that

existing solutions to the dynamic synthesis of interoperable middleware protocols do not address overall CONNECT requirements, especially missing interoperability among middleware of different types. Then, we evaluated the applicability of the aforementioned approach to application-layer interoperability at the middleware-layer concluding that the approach devised still needed some adjustments to be effective for both application- and middleware-layers interoperability.

- Since the work on application-layer mediator synthesis was based on the assumption that a model of the interaction protocol for a NS is dynamically discovered, we finally presented **a model-driven approach, based on data-flow analysis and testing, to the automated elicitation of application-layer protocols from software implementations**. The defined approach has been applied to the context of Web services and, in particular, to an existing Web service which is the Amazon E-Commerce Service. This approach presented similarities, but also several differences, with the work of work package WP4 (protocol learning). Furthermore, it allowed us to proceed in parallel with the work of WP4 and to state the requirements that the learning approaches had to satisfy to enable mediator synthesis.

1.3 First Review Recommendations and Related Reactions for Year 2

The following list of items reports the reviewers' recommendations for the work done within WP3 after the first year of the project. For each recommendation, we indicate how they are accounted for within WP3 ongoing work:

- *Recommendation 1: a means to specify the goal of the connector to be generated has to be defined.*

  Note that we so far focus on pairwise CONNECTors, i.e., the CONNECTion of two NSs, while multi-party CONNECTors involving any number of NSs is an area for future work. In that context, the specification of the CONNECTor's goal derives from the specification of the NS' *affordances*. An affordance is a high-level description of the required/provided functionalities of a NS. Semantic conceptualization of affordances using ontology is also supported to improve the performance of the matching process.

- *Recommendation 2: scalability of the synthesis process, as well as compositional connector synthesis, has to be investigated.*

  Scalability issues have not been deeply investigated yet as we have first concentrated on supporting CONNECTor synthesis, from abstract to concrete, based on the theory of mediator elicited in the first year. Scalability will be addressed during the third year of the project. However, the model abstraction step developed for the devised synthesis process (see Section 1.4) represents a first attempt to this direction by reducing the size of the behavioral models of the NSs to be CONNECTed.

  As for the previous recommendation, compositional CONNECTor synthesis will be better investigated during the third year of the project. However, the pattern-based mediator synthesis approach, which is described in Chapter 2, represents a first attempt to this direction by identifying the basic mediator patterns that, as solution to recurrent protocol mismatches, can be considered as the basic building blocks for a compositional CONNECTor synthesis method.

- *Recommendation 3: the connector synthesis technique devised after the first year of the project seems to be suitable only for centralized connectors hence following an "orchestration synthesis style". What about supporting also "choreography synthesis styles" suitable for distributed connectors?.*

The synthesized CONNECTor model can be either implemented as an additional component/system that intercepts all the interactions among the NSs in the environment or distributed as a set of wrappers. In the latter case, each wrapper may for instance be deployed locally to each NS. The wrappers cooperate with each other in order to realize the behavior expressed by the synthesized CONNECTor model.

If deadlock-freedom is taken into account by the model of the CONNECTor's goal, the synthesis of distributed CONNECTors is polynomial in the "size" of this model, hence addressing also possible scalability issues.

- *Recommendation 4: it is not clear whether distinguishing among peer-to-peer and client-server systems is significant for the connector synthesis process or not.*

  This distinction is not significant in our work since we target CONNECTion between NSs, possibly relying on heterogeneous interaction paradigms. It is then a matter of considering a notion of matching based on either simulation (for client-server systems) or bisimulation (for peer-to-peer systems).

1.4 Overview of the Unified CONNECTor Synthesis Process

With the recommendations from the first review in mind and by going a step forward with respect to the results achieved in Year 1 towards meeting WP3 objectives (see Section 1.1), the main challenge for Year 2, has been to: **deliver a unified process, and related artifacts, for the automated synthesis of mediators at both application and middleware layer**. We summarize below the resulting process while the development of its constituents is detailed in the following chapters.

Starting from two protocols[1], P and Q, which differently implement similar functionalities, we want –if possible– to automatically synthesize at run-time, a mediator that makes the two protocols able to interoperate. This is achieved in the following 5-steps process. For each process step, we refer to the chapters that describe the theories, models, techniques, and tools defined -or used- to realize the step.

Step 1: Middleware abstraction (Chapter 4). As shown in Figure 1.2, *Step 1* starts from the abstract specification of the interaction behavior (or protocol) of the NSs, which is given as a *Labeled Transition System* (LTS) [47] and is obtained via the Discovery enabler, possibly using the Learning enabler. The given model of the interaction behavior embeds middleware-specific information that partly characterizes the semantics of the communication. Such information is abstracted in this step using reference high-level communication actions to reason about the CONNECTion of NSs despite possible heterogeneity in the communication paradigms they use, as detailed in Chapter 4. This leads to produce middleware-agnostic LTSs for further processing in Step 2. The applied middleware abstraction rules are stored in order to be reused backwards when performing *concretization* (see *Step 4*).

Step 2: Common abstractions (Chapter 4). According to the theory of mediators introduced in previous year deliverable and its revision presented in Chapter 3, the objective of *Step 2* is to check whether the middleware-agnostic LTSs associated with the NSs to be CONNECTed can share the same alphabet, at a certain level of abstraction. This step exploits the specification of the NSs' observable actions using ontology and hence the related semantics knowledge, which allows identifying mapping between NSs actions. As a result, abstract LTSs that refer to common actions are produced from the middleware-agnostic LTSs (see Figure 1.3).

Step 1 and *Step 2* allow for reasoning on more abstract models of the NSs interaction behavior hence addressing, to some extent, scalability issues.

[1] As mentioned, we limit the number of protocols to two only as first step. Still, the discussed process can always be generalized to an arbitrary number of protocols.

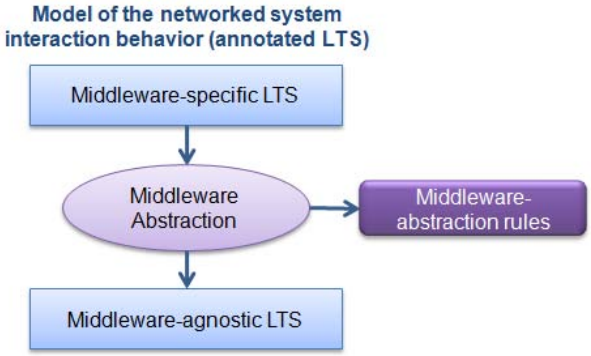


Figure 1.2: Elicitation of middleware-agnostic protocols

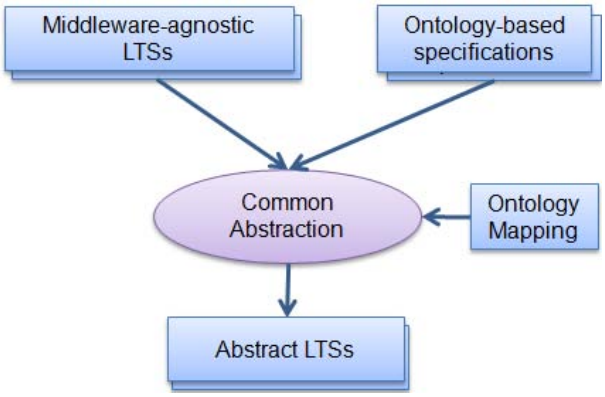


Figure 1.3: Abstracting protocol actions

Step 3: Abstract CONNECTor synthesis (Chapters 3 and 4). As shown in Figure 1.4 and in accordance with the proposed theory of mediators, *Step 3* relies on two relations defined over the abstract LTSs produced at Step 2, which abstract the interaction behavior of the NSs to be CONNECTed: (i) *functional matching* and (ii) *protocol mapping*. Functional matching serves to identify the existence of common traces that lead the NSs to coordinate and hence to achieve a common goal (i.e., the existence of a CONNECTor). Subsequently, if at least one common trace has been found (and it leads to achieve the goal specified by the required/provided affordance model), the mapping between the two LTSs (i.e., protocol mapping), over the common traces, is automatically computed, hence producing an abstract model of the CONNECTor's interaction behavior.

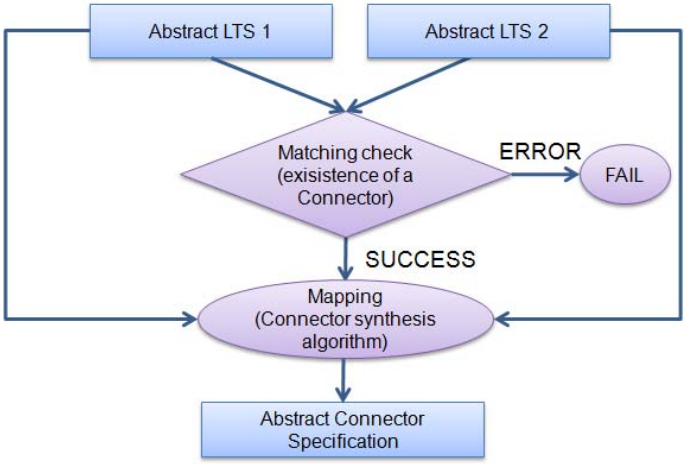


Figure 1.4: Existence and subsequent synthesis of a CONNECTor

Step 4: From abstract to concrete CONNECTor (Chapter 5). Given the abstract specification of the mediator behavior synthesized at Step 3, and as depicted in Figure 1.5, *Step 4* generates a concrete CONNECTor. In effect, *Step 4* actually merges, with the mediator behavior, the abstractions associated with Step 1 (i.e., Middleware-abstraction rules) and reverses them to apply concretizations of NS actions based on the ontology-based action mapping associated with Step 2 (i.e., Ontology Mapping).

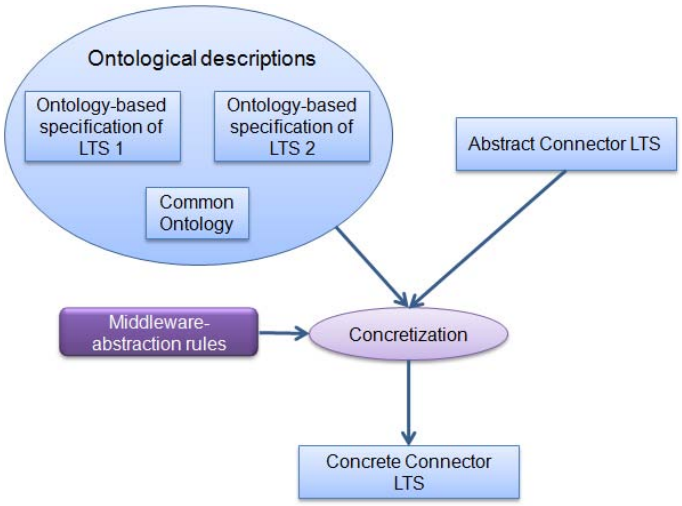


Figure 1.5: Concrete CONNECTor generation

Step 5: Concrete CONNECTor code generation (Chapter 5). By referring to Figure 1.6, the *Step 5* automatically generates the actual code implementing the concrete CONNECTor produced at Step 4. The CONNECTor implementation in particular relies on actual communication with the NSs, which is achieved using enablers (i.e., listeners and actuators) developed within WP1. Step 5 further exploits Model-To-Text transformation techniques. For instance, it makes use but not limited to, JET for a Java-based CONNECTor implementation.

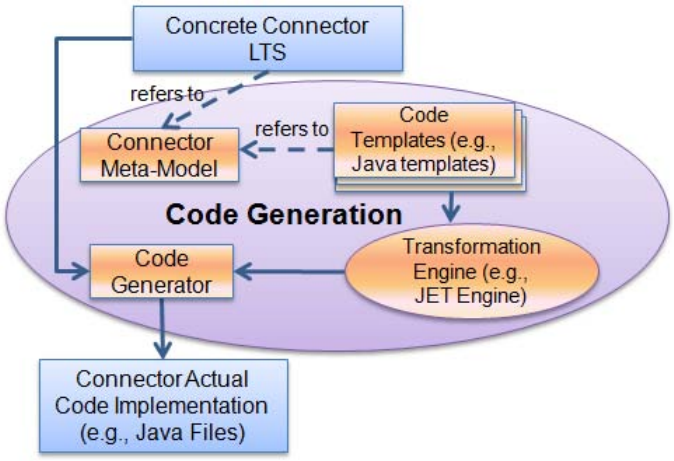


Figure 1.6: Code generation

The five steps of the CONNECTor synthesis process shall account for both the functional and non-functional behavior of CONNECTors. However, as already mentioned, work of the 2nd year has primarily focused on the functional dimension, while non-functional properties will be considered in the 3rd year, in particular building upon latest results of WP5 (Chapter 6 describes our initial effort in this direction).

1.5 Illustrative Example: the Photo Sharing Scenario

In order to illustrate our approach to CONNECTor synthesis that is detailed in the next chapters, we consider the simple, yet challenging common scenario of Photo Sharing within a public space such as a stadium which is illustrated in Figure 1.7.



Figure 1.7: Overview of the Photo Sharing scenario

Typically, the target environment allows for both Infrastructure Based (IB) and ad hoc 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 the handheld. Moreover, while the Photo Sharing functionality is present in both versions of the Photo Sharing application, it is unlikely that they feature the very same interface and behavior. In particular, the RPC interaction paradigm suits quite well the 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.

Typically, the case we are addressing in this scenario deals with NSs that are willing to communicate to reach a common goal, specified by their required/provided affordance model, but cannot communicate because of protocol mismatches. Our approach allows the dynamic creation of an adaptation protocol (i.e., mediator) among the NSs that is able to properly mediate their communication.

1.6 Challenges for Year 2 and Overview of the Related Achievements

As discussed in previous sections, our work within WP3 during the second year has been on eliciting a comprehensive CONNECTor synthesis process together with supporting methods and tools, based on the theory of mediator introduced in Deliverable D3.1 [5]. In particular, our work has led us **to develop automated techniques and tools to support the phases of the devised synthesis process**, from functional matching and protocol mapping, to CONNECTor model synthesis and CONNECTor code generation. We have further initiated work on **the integration of QoS management in the CONNECTor synthesis process**.

In more detail, the challenges that have been addressed during Year 2, and the related achievements, include the following, as detailed in the next chapters:

- **Approaching the CONNECTor synthesis problem in a systematic way by adopting a pattern-based solution.** As detailed in Chapter 2, we have characterized the *protocol mismatches* that we intend to solve with our CONNECTor synthesis process, as well as the basic *mediator patterns* that solve the classified problems. We believe that this classification of mediator patterns can serve in Year 3, up to the end of the project, as a basis for addressing compositional CONNECTor synthesis, which has not been addressed so far. Hence, the work informs not only WP3 but also the definition of the CONNECTor algebra in WP2.

- **Revising and extending the theory of mediators.** In Year 1, we elaborated a theory of mediators, which defined the associated matching and mapping relations over the interaction behaviors of NSs abstracted as LTSs. During Year 2, we have revised the definition of the theory, which enables us to introduce simpler definition of protocol matching and mapping. The resulting adaptation of the theory is presented in Chapter 3. Referring to the CONNECTor synthesis process (Section 1.4), the proposed theory informs the design of Steps 2 and 3.

- **From theory to abstract CONNECTor synthesis.** While our theory of mediators is defined over protocols defined in terms of highly abstract observable actions, actual CONNECTor synthesis requires dealing with the protocols that are executed by the NSs, which rely on communication actions offered by the underlying middleware. Hence, the semantics of the

protocols' observable actions must account for the semantics of actions at both application- and middleware-layers. In addition, the CONNECTor synthesis should be efficient so that it is tractable at run-time, hence leading to more constrained matching and mapping relations than those of Chapter 3. Such issues are addressed in Chapter 4, which effectively deals with the implementation of Steps 1 to 3, while leveraging our theory of mediators.

- **From abstract to concrete CONNECTor deployment.** Translating the synthesized CONNECTor model into an executable artifact that can be deployed and run requires devising the runtime architecture of CONNECTors; related to this is the issue of generation of code versus interpretation of the CONNECTor model. Chapter 5 specifically addresses the CONNECTor code generation issues, and goes from the construction to the deployment and the starting up of the CONNECTor artifacts, hence dealing with Steps 4 and 5 of the synthesis process.
- **Taking into account non-functional interoperability.** While Chapters 2 to 5 deal with CONNECTor synthesis from the functional standpoint (i.e., allowing NSs to be CONNECTed), the CONNECTability of systems has further to deal with the quality of the CONNECTion (i.e., *how* it should be provided). Indeed, while building an interoperability solution, both functional and non-functional properties of the CONNECTed system under-construction must be taken into account and ensured. Chapter 6 discusses preliminary efforts towards this direction, building upon WP5 results.

As detailed in the following, significant progress has been achieved in WP3 towards actually enabling the CONNECTion of NSs despite heterogeneity, from the application down to the middleware layers. This is further accompanied by the development of supporting tools so as to experiment with the proposed approach, as in particular investigated in WP1 through integration into related CONNECT enablers. The software tools developed during Year 2 as companion prototypes of this deliverable are reported in [6].

Still, further development is needed to turn the overall theory into practice, as sketched in Chapter 7.

2 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 [8, 23, 10, 37]. Indeed, in this chapter, which is an extended and revised version of the preliminary work discussed in [74], 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 by CONNECT , and thus accounted for in the definition of *functional matching* and *protocol mapping* used in Steps 2 and 3 of the synthesis process presented in the introduction.

The design building blocks that we present include:

1. An Architectural Pattern called *Mediating Connector*, which is the key enabler for communication;

2. A set of *Basic Mediator Patterns*, which describe: (i) the 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 1.5, for which multiple versions of both the IB and P2P implementations may be envisioned. 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 2.1 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 name of actions are self-explanatory. 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 2.2, 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 behaviors. This second difference applies also if one considers two different versions (among the four) of the P2P implementation. Indeed, two instances of the same version (e.g., P2P Photo Sharing version 1) are compatible and hence are able to interoperate while two instances of different versions (e.g., Version 1 and Version 3) are not. 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 2.2 a)) and the P2P Photo Sharing Version 1 (Figure 2.1 v1)).

In this chapter, we make some assumptions and we investigate the related underling research problems as part of CONNECT . 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 [40, 17]. We also assume a semantic correspondence between the messages exchanged among components exploiting ontologies.

The chapter is organized as follows. Sections 2.1 and 2.2 respectively define the *Mediating Connector Architectural Pattern* and the related *Basic Mediator Patterns*. Then, Section 2.3 illustrates the application of the mediator patterns to the Photo Sharing scenario by showing how the defined patterns can be used to solve interoperability mismatches. Finally, Sections 2.4 and 2.5 respectively discusses related work and concludes by a summary of the chapter contribution and perspective for future work.

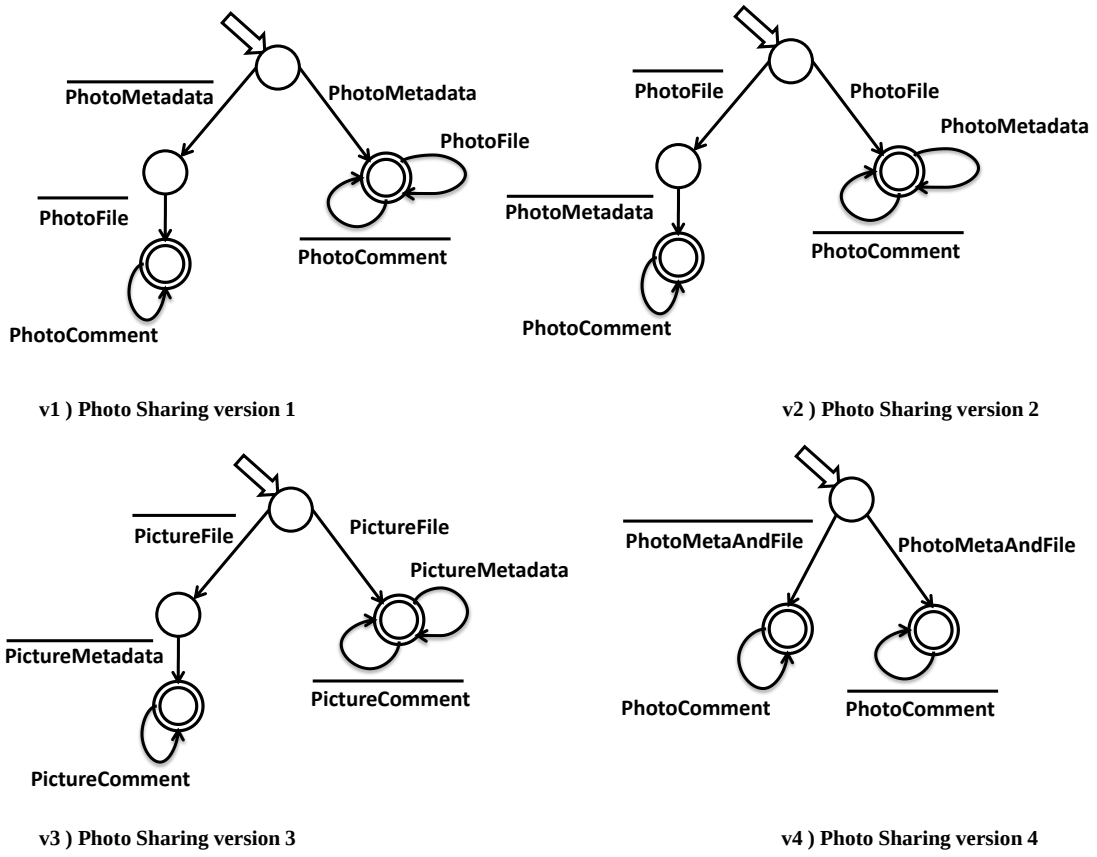


Figure 2.1: Peer-to-Peer-based (P2P) implementation

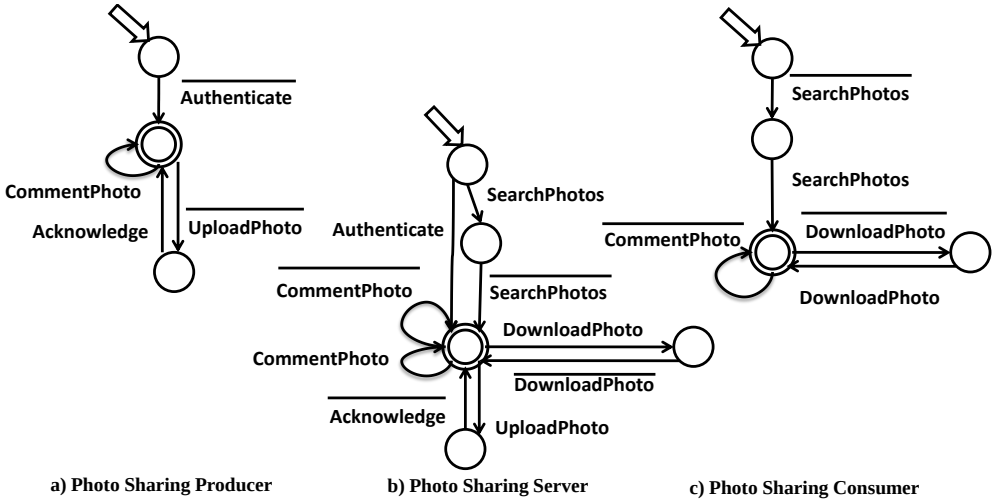


Figure 2.2: Infrastructure-based (IB) implementation

2.1 Mediating Connector Architectural Pattern

The interoperability problem between diverse components populating the pervasive networking environment and its related solution is characterized as a *Mediating Connector Architectural Pattern* based on the template used in [23] 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 behavioral pattern and represents the architectural building block embedding the necessary support to dynamically cope with components' behavioral diversity.

Name. Mediating Connector.

Also Known As. Mediator.

Example. Consider the pervasive networking environment that embeds networked devices from a multitude of applications domain, for example consumer electronics or mobile and personal computing devices. Suppose that *potentially compatible* applications running on various devices want to *interoperate*. Potentially compatible applications are applications that may share some *intent* resulting in complementary portions of their *interaction protocols*, i.e., complementary sequences of messages visible at interface level. In principle, those applications should be able to interoperate, but because of some *behavioral differences* that they exhibit, they are not compatible. With interoperate we mean *coordinate* and *communicate* (i.e., *synchronize*). For example, consider the Photo Sharing applications: users of different applications may want to communicate and in principle this should be possible since the different applications implement similar functionalities. However they do it in different ways and this prevents the communication. For instance, the infrastructure-based producer (Figure 2.2 a)), may want to communicate with the peer-to-peer Photo Sharing version 1 (Figure 2.1 v1)) (or also with version 2, 3, or 4 - Figure 2.1 v2), Figure 2.1 v3) or Figure 2.1 v4) respectively), and in principle this should be possible. Nevertheless, their behavioral mismatches prevent the communication.

Context. The considered environment is pervasive, that is distributed and continuously changing due to dynamically appearing and disappearing systems. Although such systems populating the environment are heterogeneous (mismatching) they can potentially interoperate and hence 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 [23]) 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 2.2 on *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 behavioral differences between potentially 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 manage the behavioral 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 opportune translation/adaptation of protocols when/if needed.

Structure. The Mediating Connector Pattern comprises three types of participating components: communicating components, potentially compatible components and mediators.
The *communicating components (or applications)* implement already compatible components, i.e., components able to interact and evolve following their usual interaction behavior. The *potentially compatible components (or applications)* implement the application level entities (whose behavior, 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 potentially compatible components can only evolve following their usual interaction behavior, 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 potentially compatible components that want to interoperate. Figure 2.3 shows the components involved in a mediated system.

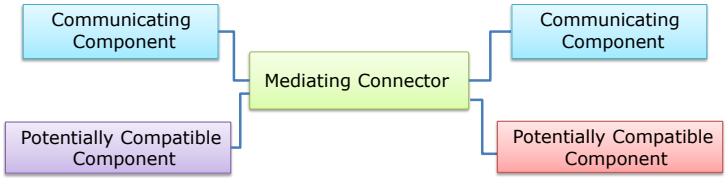


Figure 2.3: Entities involved in a mediated system

Dynamics. The dynamics refers to the interactions between components (applications) and mediators. Triggered by a user, the IB producer protocol (Figure 2.2 a) performs one of its possible behaviors: it authenticates and then uploads one photo receiving the corresponding 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, which are communicating components, (2) manipulate/translate and forward the upload and acknowledge messages between the IB producer (Figure 2.2 a) and the P2P Photo Sharing version 1 protocol (Figure 2.2 v1)), which are potentially compatible components. With the term "translation" we mean not just a language translation but also a "behavioral translation" (see Section *Basic Mediator Patterns* for details).

Implementation. The implementation of this pattern implies the definition of an approach/tool to automatically synthesize the behavior of the Mediating Connector, which allows the potentially compatible components to interoperate by mediating their interactions.

Example Resolved. The Mediating Connector's concrete protocol for our running example is shown in Figure 2.10. Once established that they are potentially compatible (i.e., they have some complementary portion of interaction protocols), the mediating connector manages the components' behavioral 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 behavior.

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 behavioral 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, are not able at all to interoperate.

2.2 Basic Mediator Patterns

Following the characterization of the Mediating Connector pattern, 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 [62]. Figure 2.4 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 potentially compatible components. All the considered traces are the most elementary with respect to the messages exchanged and only visible messages are shown.

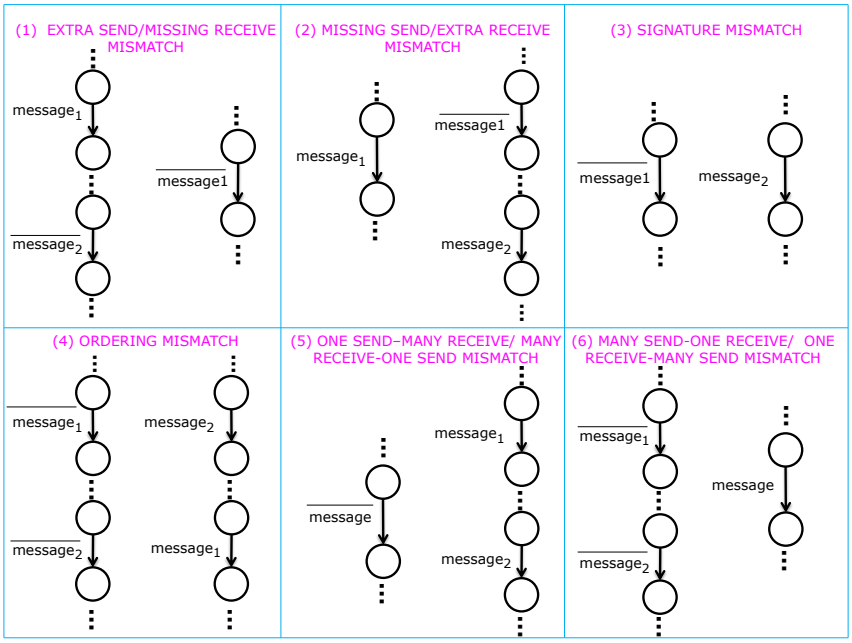


Figure 2.4: Basic interoperability mismatches

It is obvious that, in real cases, the traces may also contain portions of behavior 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 behaviors) illustrated in Figure 2.5 where only their visible messages are shown (messages that they exchange with the components).

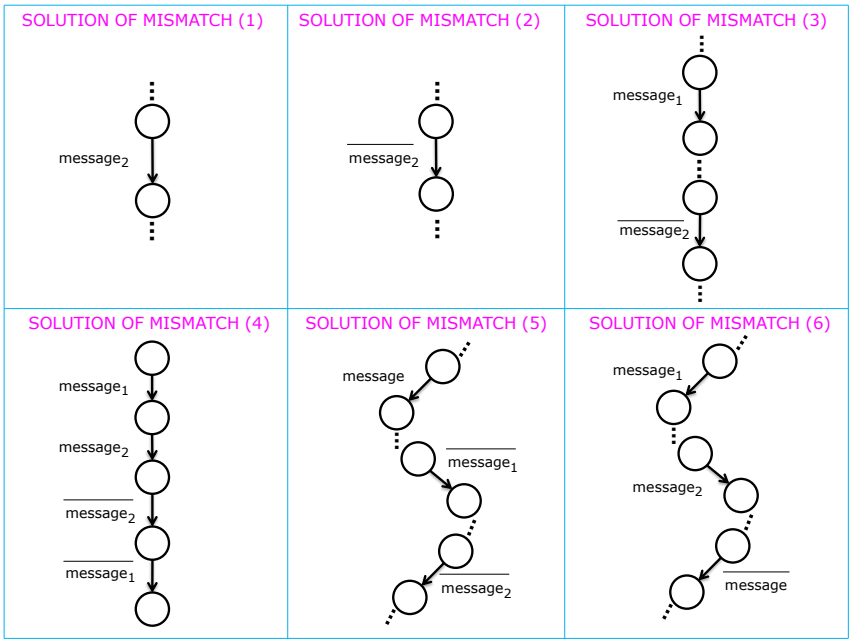


Figure 2.5: Basic solutions for the basic mismatches

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, which otherwise would not be possible because of behavioral mismatches.

(1) MESSAGE CONSUMER PATTERN.

Problem. (1) Extra send/missing receive mismatch ((1) in Figure 2.4, where the extra send action is $\overline{message_2}$). One of the two considered traces either contains an extra send action or a receive action is missing.

Example. Consider two traces implementing the abstract action "upload photo (respectively download photo)". For example, in the mismatch (1) of Figure 2.4 the right trace implements only the sending of the photo ($\overline{message_1}$) while the left trace implements the 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 2.5) 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 2.6 and are:

(a) $message_1$ has exchanged 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 $message_1$.

(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 $message_1$.

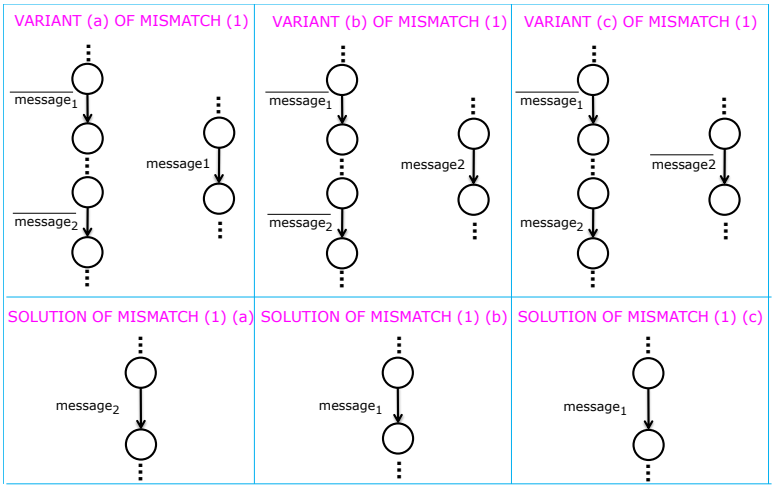


Figure 2.6: Variants of the Basic Mediator Pattern (1)

(2) MESSAGE PRODUCER PATTERN.

Problem. (2) Missing send/extra receive mismatch ((2) in Figure 2.4, 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 2.4, 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 of the message ($message_1$).

Solution. Introducing a *message producer* (solution of mismatch (2) in Figure 2.5) 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 2.7 and are:

(a) $message_1$ has exchanged 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 $\overline{message_1}$.

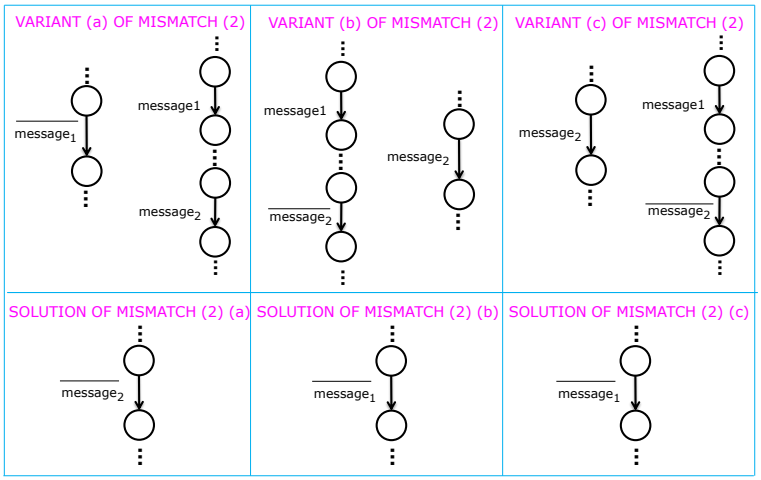


Figure 2.7: Variants of the Basic Mediator Pattern (2)

(3) MESSAGE TRANSLATOR PATTERN.

Problem. (3) Signature mismatch (upper right box of Figure 2.4). 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 2.4, $\overline{message_1}$ could be the sending of a message *PhotoFile* while $message_2$ the receiving of a *PictureFile* message.

Solution. Introducing a *message translator* (solution of mismatch (3) in Figure 2.5). It receives the request and sends it after a proper translation. We assume the existence of some entity able to do the 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 2.8 and amount to exchange sender/receiver roles between the two traces, i.e., $message_1$ and $\overline{message_2}$ and the solution is

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

made by $message_2 \,.\, \overline{message_1}$.

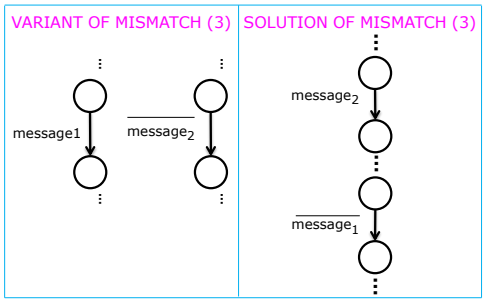


Figure 2.8: Variants of the Basic Mediator Pattern (3)

(4) MESSAGES ORDERING PATTERN.

Problem. (4) Ordering mismatch ((4) in Figure 2.4, 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 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.

Example. Consider two traces implementing the abstract action "send (respectively receive) photo". $message_1$ and $message_2$ in the mismatch (4) of Figure 2.4, 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 2.5). This pattern has a compatible behavior for both 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 variants and respective solutions are shown in Figure 2.9 and are:

(a) left trace has exchanged 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 $\overline{message_2}.\ message_1$.

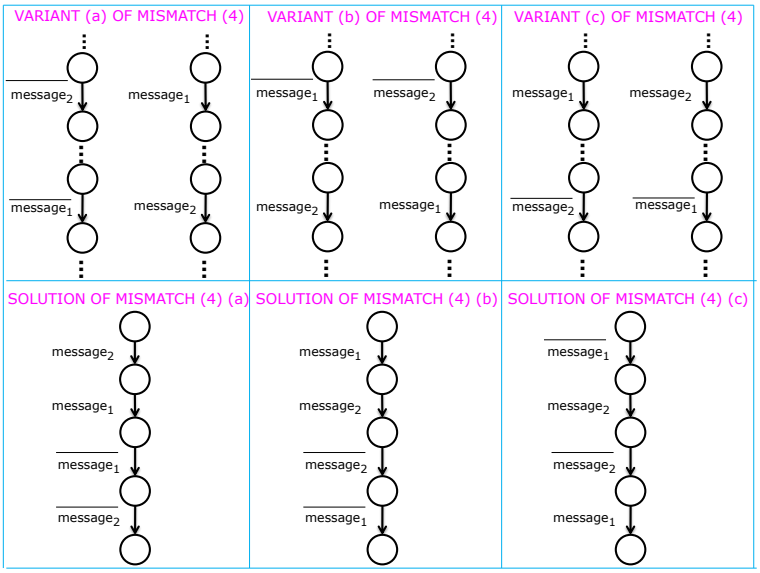


Figure 2.9: Variants of the Basic Mediator Pattern (4)

(5) MESSAGE SPLITTING PATTERN.

Problem. (5) One send-many receive/many receive-one send mismatch ((5) in Figure 2.4). 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 2.4, 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 2.5). It receives one message from one side, splits it properly, and sends the split messages to the other. We assume the existence of some entity able to do the splitting operation[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.

Problem. (6) Many send-one receive/one receive-many send mismatch ((6) in Figure 2.4). 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 2.4, for example, $\overline{message_1}$ and

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

$\overline{message_2}$ are the sending of two separate messages $\overline{PhotoMetadata}$ and $\overline{PhotoFile}$ while $message$ is the receiving of $PhotoMetaAndFile$.

Solution. Introducing a *messages merger* (solution of mismatch (6) in Figure 2.5). It receives two messages from one side, merges them properly, and sends the merged messages to the other. We assume the existence of some entity able to do the merge operation. 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, $PhotoMetadata\ .\ PhotoFile\ .\ \overline{PhotoMetaAndFile}$.

2.3 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 some *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 2.2 a)) and the P2P Photo Sharing version 1 (Figure 2.1 v1))) are potentially 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.

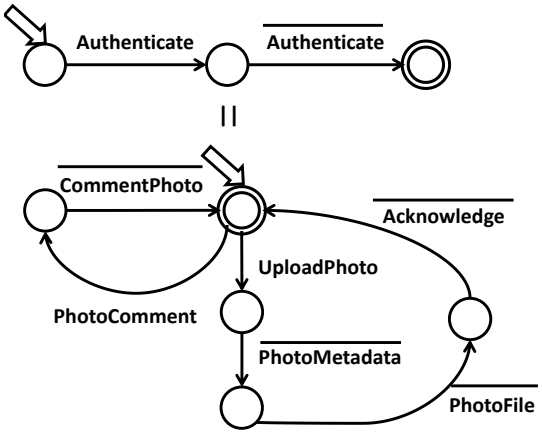


Figure 2.10: Behavioral description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 2.2 a) and P2P Photo Sharing version 1 of Figure 2.1 v1))

Figure 2.10 shows the behavior of the Mediating Connector for the applications considered in our example. We recall (as already sketched in the beginning of Chapter 2) 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 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.

2.4 Related Work

In the last decades, protocol mediation has been investigated in several contexts, among which design patterns [37]. 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 [8]: "*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, Bushmann *et al*. [23] 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 [10], which proposes a pattern language. The "gang of four" in [37] 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, 29, 50, 43]. In particular, references [50, 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 [50], 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.

Finally, reference [33] 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.

2.5 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.

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 3 and 4 that respectively present a theory of mediators and its application to solve such kind of interoperability mismatches.

An interesting perspective for future work is to exploit the algebra of connector presented in Deliverable D2.2 [4] in order to implement a compositional approach based on patterns, which allows: the component's behavior decomposition, the reasoning on mismatches, and the synthesis of mediating CONNECTor behavior.

Moreover, in the direction of automated code generation, it would be of interest to provide the "concrete" Basic Mediator Patterns, i.e., the skeleton code corresponding to the "abstract" patterns presented in this chapter.

3 A Theory of Mediators for Eternal Connectors

In our work, we want to approach the protocol interoperability problem in an automated way. Then, the solution we address here, is to automatically synthesize *mediators* that allow protocols to interoperate by solving their behavioral mismatches. While in Chapter 2, 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 2). Still, among all the implementations that are sketched, we consider as reference scenario, the one made by: the IB Photo-Sharing Producer (Figure 2.2 a)) and the P2P Photo-Sharing version 1 (Figure2.1 v1)).

This chapter specifically presents a *theory of mediators* for CONNECTors, which is a revised and extended version of the supporting theory introduced in Deliverable D3.1 [5]. After the results achieved during the first year of research within the CONNECT project, a simpler way of abstracting behaviors, and thus of matching, has been found (as mentioned during the first review). The main differences between the previous formal approach and the theory of mediators presented in this deliverable are:

(i) **The abstraction strategy used in the synthesis process:** In the first version, the abstraction was a two-steps process. Given two protocols P and Q, the process included: (1) finding the so called *structure LTSs* of P (and of Q respectively), which is a more abstract LTS preserving some *structural characteristics* of P (and of Q respectively), i.e., *rich states* that are states identifying branches, entry points for cycles, and joins; and (2) finding the *induced LTSs* of P (and of Q respectively) by applying some ontological knowledge on both the obtained structure LTSs of P and Q. Instead, in the current theory, the abstraction is a one-step process, which is only based on ontological knowledge about the protocols that is exploited against their LTS models in order to obtain their abstractions.

(ii) **The matching (compatibility check) between protocols:** The protocol matching is strictly related to the abstraction method. Then, in the first version of the theory, the check was "structure-based", i.e., it was an equivalence check based on a suitable notion of bisimulation (or simulation). In the current version, instead, the check is based on a notion of trace equivalence having relaxed the structural constraints.

Consequently, we have revised and changed the overall theory formalization as shown in the following. The theory of *mediators* characterizes:

* **The interaction protocols of networked systems that are functionally matching but behaviorally mismatching:** We assume that the specification of the protocols is provided, either as part of the advertisement of networked systems using some discovery protocol or based on some learning technique, as developed in WP4, like the one discussed in [40].

* **The interoperability notion between protocols based on functional matching:** Note that in a first step, we restrict ourselves to interoperability between pairs of protocols and we further do not explicitly address data heterogeneity, which is being extensively addressed elsewhere [63]. Some data-level heterogeneity is approached within Chapter 4 and we plan to investigate other levels of data-heterogeneity in the future.

* **The behavior of mediators to achieve interoperability under functional matching despite behavioral mismatches**.

This chapter is organized as follows. Section 3.1 sets the principles of our approach, including related terminology. Then, Sections 3.2 to 3.4 introduce a formalization for interaction protocols, which paves the way for automated reasoning about protocols functional matching and for the automated synthesis of mediators. Finally, Section 3.6 positions the chapter's contribution with respect to related work, while Section 3.7 draws some conclusions.

3.1 Towards Emergent Mediators

The focus of this chapter is the *protocol interoperability problem* and our goal is to find an *automated solution* to solve it dynamically. We give below the necessary definitions that set the context of the work described in this chapter:

- With the term *protocols*, we refer to *application-layer interaction protocols* or *observable protocols*. That is, a protocol is the behaviour of a system in terms of the sequence of messages visible at the interface level, which it exchanges with other systems.
- We further focus on *compatible* or *functionally matching* protocols. Functionally matching means that protocols can *potentially communicate* by performing *complementary sequences of actions*. *Potentially* means that communication may not be achieved because: (i) the languages of the two protocols are different/discrepant/mismatching, although semantically equivalent; or (ii) the sequence of actions performed by a protocol is different from the sequence of actions of the other one because of interleaved actions related to third parties communications (i.e., exchanged with other systems, the environment). In the former case, it is necessary to properly perform a manipulation of the two languages. Note that this case relates to basic mediator patterns introduced in the previous chapter. In the latter case, it is necessary to provide an abstraction of the two 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 abstracted sequences of actions are complementary, i.e., are the same sequences of actions while having opposite send/receive (output/input) "type" for all actions.
- With *interoperability*, we mean the property referring to the ability of heterogeneous protocols that *functionally match to coordinate* where the coordination is expressed as synchronization, i.e., two systems succeed in coordinating if they are able to synchronize.

Summarizing, we want to achieve automated and on-the-fly interoperability between behaviorally mismatching, yet functionally matching application-layer interaction protocols.

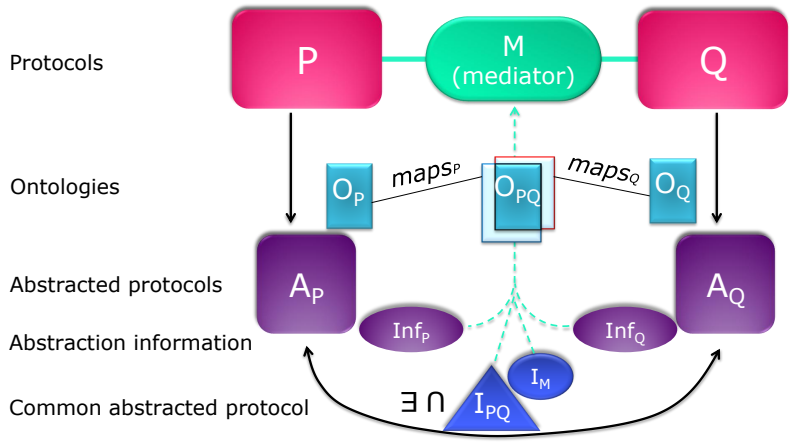


Figure 3.1: An overview of our approach

Figure 3.1 depicts the main elements of our methodology towards this goal:

(i) Two application-layer protocols P and Q whose representation is given in terms of *Labeled Transition Systems* (*LTS*s) [47], 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. 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 the relabeling of P and Q, respectively, where the actions not belonging to the common language O_{PQ} are hidden by means of silent actions (τs); moreover, we store some abstraction information (exploited in order 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) After, we check the compatibility of the protocols by looking if there exist complementary traces (the set I_{PQ} in figure), modulo mismatches and third parties communications, in the set 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 these matching information (I_M) that will be exploited during the mapping.

(vi) Finally, given two protocols P and Q, and an environment E, 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.

Hence, referring to the CONNECTor synthesis process of Chapter 1, the proposed theory of mediators tackles Steps 2 and 3, and Step 1 to some extent (considering that the ontologies characterize also middleware-layer communication actions).

3.2 A Formalization of Protocols

As discussed previously, a protocol is the behavior of a system in terms of the actions it exchanges with its environment, i.e., other protocols. We further exploit LTS to characterize such behavior.

LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioral 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 usual convention that for all* $l \in L, l = \bar{\bar{l}}$.
- $D \subseteq S \times L \times S$ *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 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 behaviors 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 $t = l_1, l_2, \ldots, l_n \in L^*$ is such that:*

$$\exists(s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ \ldots \ s_m \xrightarrow{l_n} s_n) \text{ where } \{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F.$$

We also 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 sequence in a protocol (it may be also a trace). More formally:

Definition 3 (*Subtrace*) *Let $P = (S, L, D, F, s_0)$. A subtrace $st = l_i, l_{i+1}, \ldots, l_n \in L^*$ is such that:*

$$\exists(s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2} \ \ldots \ s_m \xrightarrow{l_m} s_n) \text{ where } \{s_i, s_{i+1}, s_{i+2}, \ldots, s_m, s_n\} \in S$$

We adopt the notion of parallel composition *à la* CSP [72]. We recall that the semantics of the parallel composition is that processes P and Q need to synchronize on common actions, while they can proceed independently when engaged in non common actions.

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 between 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:*

$$\frac{P \xrightarrow{m} P'}{P||Q \xrightarrow{m} P'||Q} \qquad m \notin L_Q$$

$$\frac{Q \xrightarrow{m} Q'}{P||Q \xrightarrow{m} P||Q'} \qquad m \notin L_P$$

$$\frac{P \xrightarrow{m} P'; Q \xrightarrow{\overline{m}} Q'}{P||Q \xrightarrow{\tau} P'||Q'} \qquad m \in L_P \bigcap L_Q$$

Note that when we build the parallel composition of protocols P and Q with the environment E and the mediator M, the composed protocol is restricted to the languages of P and Q thus forcing them to synchronize.

3.3 Abstract Protocol

Given the definition of extended LTS associated with two interaction protocols run by networked systems, 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, ontology mapping functions $maps_P$ on P and $maps_Q$ on Q, and their intersecting common ontology O_{PQ}, there exists *at least one complementary trace* (one in P and one in Q) that allows P and Q to coordinate. 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. Thus, we expect to find, at a given level of abstraction, a common protocol that represents the potential interactions of P and Q. This leads us to formally analyze such alike protocols to find –if it exists– a suitable mediator that allows the interoperability that otherwise would not be possible.

In order to find the protocols' abstractions, we exploit the information contained in the ontology mapping to suitably relabel the protocols. We consider the two ontologies mapping functions to the common ontology as given. Specifically, as detailed in the following, the relabelling of LTSs produces new LTSs that are labelled only by common actions and τs, and hence are more abstract than before (e.g., sequences of actions may have been compressed into single actions). For illustration, Figure 3.2 summarizes the ontological information of the IB Producer of Figure 2.2 a)(first column) and of the P2P Photo-Sharing version 1 of Figure2.1 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.

Infrastructure-based Photo-Sharing Producer	**Common 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 3.2: Ontology mapping and common language between Infrastructure-based Photo-Sharing Producer and the peer-to-peer Photo-Sharing version 1 (Figure 2.2 a) and Figure2.1 v1) respectively)

We specialize the usual ontology mapping definition [45, 46] by considering pairs of elements made by more than one label. We use such specialized ontology mapping on protocols' ontology where the vocabulary of the source ontology is represented by the language of the protocol. The ontology mapping is the result of the application of an ontology mapping function $maps$ on a protocol. This application maps P's ontology into another ontology O. More formally:

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$,
- $O_P = (L_P^*, A_P)$ *be the ontology of* P *where* L_P^* *is the vocabulary and* A_P *are the axioms,*
- $O = (L, A)$ *be an ontology,*
- $maps : L^* \rightarrow L$ *be an ontology mapping function,*
- $st \in L_P^*$ *be a subtrace on* P.

The ontology mapping is: $\{l \in L : l = maps(st)\}$

By applying the above ontology mapping, we relabel protocols with labels of their common language and τs for the thirds parties languages. To identify the common language, we first map each protocol's ontology into a common oontology and then by intersection, we find their common language.

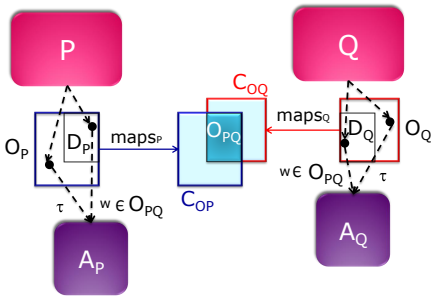


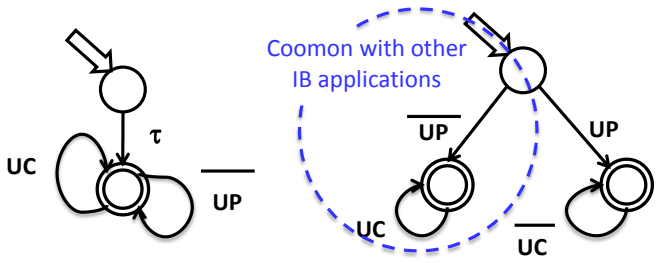
Figure 3.3: Abstract protocol building

Figure 3.3 depicts the abstraction of the protocols. Let us consider two protocols P and Q with respective ontologies O_P and O_Q and ontology mapping functions $maps_P$ and $maps_Q$. We first use the mapping functions to map O_P and O_Q into a target ontology where C_{OP} and C_{OQ} represent the codomain sets of $maps_P$ and $maps_Q$ respectively. The subsets D_P and D_Q of O_P and O_Q, respectively, represent the portion of the domains of $maps_P$ and $maps_Q$ respectively corresponding to C_{OP} and C_{OQ}. Note that we consider protocols such that for each element of the codomain corresponds only one element of the domain.

The common language between P and Q is defined as the intersection O_{PQ} of C_{OP} and C_{OQ}. In particular, the actions belonging to the common language are pairs of actions with opposite type send/receive (output/input). The actions in the pairs result from the ontology mapping of actions belonging to P and Q respectively. For instance, the pair ($\overline{UP}$, UP), i.e., the first row in the example of Figure 3.2, is in the common language being made up by the result of the ontology mapping on the IB application and on the P2P application respectively. We define below the common language more formally:

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and let* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$,
- t_P, t_Q *be traces of* P *of* Q *respectively,*
- $O_P = (L_P^*, A_P)$ *be the ontology of* P *and let* $O_Q = (L_Q^*, A_Q)$ *be the ontology of* Q,
- $O = (L, A)$ *be an ontology,*
- $maps_P : L_P^* \rightarrow L$ *be the ontology mapping function of* P *and let* $maps_Q : L_Q^* \rightarrow L$ *be the ontology mapping function of* Q, *and*
- $l = maps_P(t_P)$ *and let* $l' = maps_Q(t_Q)$ *(hence* $l \in L$ *and* $l' \in L$*).*



(a) IB Producer (b) P2P Photo Sharing version 1

Figure 3.4: Abstracted LTSs of the Photo Sharing protocols

The common language O_{PQ} between P and Q is defined as:

$$O_{PQ} = \{(l, l') : l = \overline{a} \wedge l' = a \ (resp. l = a \wedge l' = \overline{a}) \ \}$$

where t_P, t_Q are arbitrary sequences of actions possibly implementing a basic mismatch (as defined in Section 2.2 –see also Figure 2.4)

The relabeled protocols A_P and A_Q, abstracted from P and Q respectively, are built as follows:

1. The chunks (sequences of states and transitions) of P and Q labelled by traces on D_P and D_Q, respectively are substituted by building a single transition labeled with a label on O_{PQ};
2. All the other transitions labelled with actions belonging to the thirds parties language, are relabelled with τs .

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

Definition 7 (*Relabeling 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 *and* O_Q *be ontologies of* P *and* Q *respectively,*
- $maps_P$ *and* $maps_Q$ *be ontology mapping functions of* P *and* Q *respectively,*
- C_{OP} *and* C_{OQ} *be the codomain sets of* $maps_P$ *and* $maps_Q$ *respectively,*
- $D_P \subseteq O_P$ *and* $D_Q \subseteq O_Q$ *be the portions of the domains of* $maps_P$ *and* $maps_Q$ *respectively corresponding to* C_{OP} *and* C_{OQ}*,*
- O_{PQ} *be the common language between* P *and* Q*.*

The relabeling function $relabels$ *is defined as:* $relabels : (P, D_P, O_{PQ}) \rightarrow A_P$
where $A_P = (S_A, L_A, D_A, F_A, s_{0_P})$ *where* $L_A = \{l : (l, l') \in O_{PQ}\} \bigcup \{\tau\}$, $S_A \subseteq S_P$, $F_A \subseteq F_P$.
This applies similarly to Q*:* $relabels : (Q, D_Q, O_{PQ}) \rightarrow A_Q$.

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 3.4. The subsequent step is to check whether the two abstracted protocols share a *complementary coordination policy*, i.e., whether the abstracted protocols may indeed synchronize, which we check over protocol traces as discussed next.

3.4 Towards Automated Matching and Mediator Synthesis

The formalization described so far is needed to: (1) characterize the protocols and (2) abstract them into protocols on the same alphabet. 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 τs), 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 *at least one complementary trace*.

Before going into the definition of the compatibility or functional matching relation, let us first provide the one of *complementary coordination policies*. Informally, two coordination policies are complementary if and only if they are the same sequence of actions while having opposite send/receive type for all actions. That is, traces t and t' are complementary if and only if: each send action (resp. receive) of t has its complementary receive action (resp. send) in t' and similarly with exchanged 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* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$*,*
- T_P *and* T_Q *be the set of all the traces of* P *and* Q*, respectively, and*
- $t = l_1, l_2, \ldots, l_n \in T_P$ *and* $t' = l'_1, l'_2, \ldots, l'_n \in T_Q$*.*

Coordination policies t *and* t' *are complementary coordination policies iff the following conditions hold:*

(i) for each $\overline{l_i}$ $(l_i) \in t$ *there exists an* $l'_j \in t'$ *such that* $l'_j = l_i$ $(l'_j = \overline{l_i})$*;*

(ii) for each $\overline{l'_i}$ $(l'_i) \in t'$ *there exists an* $l_j \in t$ *such that* $l_j = l'_i$ $(l_j = \overline{l'_i})$*;*

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 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 the other.

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 and τs that represent third parties synchronization. We further 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 relabeling 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 at least a pair* t_i, t'_i *that modulo the* τ*s results in complementary coordination policies.*

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 shared complementary trace modulo the τs. Such third parties communications (τs) 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. 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 the 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

Then, given two protocols P and Q that functionally match, we want to synthesize a mediator M such that the parallel composition $P||M||Q$, allows P and Q to evolve to their final states. An action of P and Q can belong either to the *common language* or the *third parties language*, i.e., the environment. 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 synchronize on complementary actions. Note 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.

The mediator is 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 3.4. The second step is to check whether they share some 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 3.4. 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 2.10. In this case, only the producer has third parties language actions and then the mediator is made by the part that translates and coordinates regarding the common language and the part that simulates the environment.

3.5 Implementing the Theory

In this section we describe algorithms to abstract the protocols (Section 3.5.1), check their compatibility (Section 3.5.2) and, if possible, synthesize a mediator (Section 3.5.3). They provide an algorithmic description of the approach described in Section 3.1, and detailed in Sections 3.3, and 3.4.

3.5.1 Abstraction Algorithms

The solution we are looking for is a common protocol abstraction such that (i) makes the protocols comparable and (ii) makes it possible to synthesize a mediator that allows the communication between them. Given a protocol P, this is done by automatically producing the corresponding abstract protocol A_P. It is built by applying the mapping rules specified in O_P (that map chunks of P, or linear components, into abstract actions) hence collapsing the related transitions and states accordingly to the abstraction constraints imposed by the informal definition of chunks given in Section 3.3.

The *abstraction phase* is described by Listing 3.1. Given two minimal and deterministic protocols P_1 and P_2 and a bijective ontology mapping function $maps$ (see Definition 5), it returns two trace sets T_1' from P_1 and T_2' from P_2 on the aligned language (made up by common language and τs) and a set of pairs PS summarizing the aligned language of P_1 and P_2. This phase is made by the three sub-phases that are :

- traces extraction described by Listing 3.2,
- languages alignment described by Listing 3.3,
- traces relabelling described by Listing 3.4.

Let:
P_1, P_2, be LTSs,
T_1, T_2, T, T_1', T_2', T' be trace sets,
PS be a set of pairs of labels representing the common language and τs,
$maps$ be an ontology mapping function,
L_2 be the set of labels of P_2.

```
1  Input: minimal and deterministic LTS P1 and LTS P2, bijective ontologyMappingFunction maps
2  Output: a triple <T1', T2', PS> with types TraceSet, TraceSet, pairsSet respectively
3
4  <TraceSet,TraceSet,pairsSet> abstractionPhase(LTS P1, LTS P2, ontologyMappingFunction maps){
5      TraceSet T1,T2;
6          pairsSet PS;
7          TraceSet T1',T2';
8          T1 := extractTraces(P1);
9          T2 := extractTraces(P2);
10         // recall that P2 = (S2, L2, D2, F2, s02)
11         PS= buildCommonLanguagePlusTaus(T1, T2, L2, maps);
12         T1' := relabelTraces(T1, PS);
13         T2' := relabelTraces(T2, PS);
14         return <T1',T2', PS >;
15 }
```

Listing 3.1: Abstraction phase

The $extractTrace$ Algorithm, takes as input a minimal and deterministic LTS P and returns as output the set of all the traces of P.

```
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 (existsCoordinationPolicy(P)){
8                  T := T ⋃ extractCoordinationPolicy(P);
9          }
10         return T;
11 }
```

Listing 3.2: extractTracesAlgorithm

The **buildCommonLanguagePlusTaus algorithm** takes as input two trace sets T_1 and T_2 and associates (i) common names (also called abstract actions) to each pair of mapped (sub)traces and (ii) τs to the remaining (sub)traces. Note that the τs represent conversations exchanged with third parties and hence abstract only the actions of a protocol (the τs are not

common indeed we use subscripts to distinguish the τs of one protocol with respect to the ones of the other).

```
Input: TraceSets T_1 and T_2 (of LTSs P_1 and P_2 respectively), labelSet L_2 ,ontologyMappingFunction maps
Output: pairsSet PS

pairsSet buildCommonLanguagePlusTaus(T_1, T_2, L_2, maps) {
    pairsSet PS := empty;
        pair p_ij := empty;
    forEach trace t_i ∈ T_1{
        forEach (l_i := subtrace(t_i)) : ∃l_j ∈ L_2^* ∧ ((m_h := maps(l_i) ∧ m_h' := maps(l_j)) ∨ (m_h' := maps(l_i) ∧ m_h :=
            ↪maps(l_j))) {
            p_ij := < m_h, m_h' > or < m_h', m_h > (accordingly)
            // m_h ranges on m_1, m_2, ..., m_k;
            PS := PS ⋃ p_ij;
        }
        forEach remaining l_i := subtrace(t_i) {//third parties conversation
            p_i := < τ_h, − >
            // h ranges over 1, 2, ..., k;
            PS := PS ⋃ p_i;
        }
        forEach remaining (l_j ∈ L_2^*) {//third parties conversation
            p_j := < −, τ_h, >
            // h ranges over k + 1, k + 2, ..., k + n;
            PS := PS ⋃ p_j;
        }
      }
return PS;
}
```

Listing 3.3: Build CommonLanguagePlusTaus Algorithm

The **relabelTraces algorithm** relabels/rewrite/abstract the traces of a trace set T with other labels contained in the pairs set PS obtaining the relabelled trace sets T_1'. Given a trace $t \in T$, the relabelling substitute sequences of actions of t with the corresponding action belonging to PS. The corresponding action can be either common ranging among $m_1, m_2, \ldots, m_k$ or can be exchanged with other systems and range among $\tau_1, \tau_2, \ldots, \tau_k$

```
Input: TraceSet T, pairsSet PS
Output: TraceSet T'

TraceSet relabelTraces(TraceSet T, pairsSet PS) {
    Trace t_ir;
    forEach trace t_i ∈ T {
        t_ir := t_i;
        forEach st_j := subtrace(t_ir) : the corresponding m_i' (or m_i or τ_i) ∈ PS i.e., < m_i', m_i >∈ PS (or
            ↪< m_i, m_i' >∈ PS or < τ_i, − >∈ PS){
            t_ir := rewrite(t_ir, m_i')(or t_ir := rewrite(t_ir, m_i) or t_ir := rewrite(t_ir, τ_i)) accordingly
        }
        if t_ir ≠ t_i then
            T' := T' ⋃ t_ir;
    }
return T';
}
```

Listing 3.4: relabelTraces Algorithm

3.5.2 Matching Algorithms

The *(semantic) matching phase* is described by Listing 3.5. It checks the existence of (at least) a complementary coordination policy between two trace sets T_1' and T_2' and return an answer, the matching relation type, and the set of matching traces. The answer can be `yes` or `no`; the matching relation can be either `intersection` or `t1containst2` or `t2containst1` or `totalmatching`, in case of `answer = yes`, while in case `answer = no` the relation is the matching string `*`. This phase exploits an auxiliary procedure to perform the compatibility check between two traces which is shown by Listing 3.6.
Let:
T_1', T_2' be trace sets,
$answer$, $relation$ be strings,

$matchingTrace$ be sets of triples,
t_1, t_2 be traces.

```
1  Input: TraceSet T'_1 and TraceSet T'_2
2  Output: String answer, String relation, triplesSet matchingTraces
3
4
5  < String, String, TraceSet, TraceSet, triplesSet > matchingPhase(TraceSet T'_1, TraceSet T'_2) {
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 Boolean answer := '';
13 Boolean relation := '*';
14 triplesSet checkedTriples := emptyset; // contains triples < t_i, t_j, resp > : t_i ∈ T'_1, t_j ∈ T'_2, resp ∈ {true, false}
15 triplesSet matchingTraces := ∅;
16 for each Trace t_i ∈ T'_1 {
17     for each Trace t_j in T'_2 {
18         // element is of the kind < t_i, t_j, details_n > : t_i ∈ T'_1, t_j ∈ T'_2, details is a data structure which, among
               ↪other fields, includes resp ∈ {true, false}
19         triple element := checkCompatibility (t_i, t_j);
20         checkedTriples := checkedTriples ⋃ element;
21     }
22 }
23 matchingTraces := {< t_i, t_j, details_n >∈ checkedTriples : details_n.resp = true };
24 if (matchingTraces = ∅) then answer := NO
25 else answer := YES;
26 if (∃t'_i ∈ T'_1 : ∄ element_k = t'_i ∈ matchingTraces) then intersect1 := true;
27 if (∃t'_j ∈ T'_2 : ∄ element_k = t'_j ∈ 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 < answer, relation, matchingTraces >;
41 }
```

Listing 3.5: Matching Phase Algorithm

The $checkCompatibility$ Algorithm performs the matching check between two traces t_1 and t_2. The returned triple by the is made by: $< t_1, t_2, details >$ where t_1 and t_2 are traces and $details$ is a data structure recording the mappings between subtraces of t_1 and t_2. Among other fields it includes $resp$, ranging on $\{true, false\}$, indicating whether or not the two traces are matching.

```
1 triple checkCompatibility(Trace t_1, Trace t_2)
```

Listing 3.6: Check Compatibility Algorithm

3.5.3 Mapping Algorithms

The *mapping phase*, described by Listing 3.7, build the mediator behavior. It takes as input the answer, the relation kind and the matching traces coming from the previous phase and the pairs set (describing the common language and the third parties language) coming from the abstraction phase. The output it returns is $mediator$ which can be a non-empty LTS in case $answer$ is `yes` and is an empty LTS in case $answer$ is `no`.
Let:
$answer$, $relation$ be strings,
$matchingTrace$ be sets of triples,
PS be a set of pairs,
$mediator$ be LTSs.

```
Input: String answer, String relation, triplesSet matchingTraces, pairsSet PS (output of abstractionPhase,i.e
    ↪., buildCommonLanguagePlusTaus)
Output: LTS mediator

LTS mappingPhase (String answer, String relation, triplesSet matchingTraces, pairsSet PS) {
triple triple_aux := empty;
triple reorderedTraces := empty; // of the kind < t_i, t_j, details_n >
LTSSet medt := empty LTS set;
LTSSet media := empty LTS set;
Trace concrete_1 := '';
Trace concrete_2 := '';
if (answer = NO) then return emptyLTS;
else if (answer = YES) then {
        // matchingTraces contains triples < t_i, t_j, details_n >
        forEach triple_i ∈ matchingTraces
    triple_aux := triple_i;
        if (checkToReorder(triple_aux) = true) then
           reorderedTraces := reorder(triple_aux);
        forEach next action act of both t_i, t_j ∈ triple_aux {
            if (act := subtrace(t_i) ∧ act = τ_h ∧ t_i ∈ triple_i ∧ act := subtrace(t_j) ∧ act := τ_k ∧ t_j ∈ triple_i) then {
                concrete_1 := concretizeAction(τ_h, PS);
                concrete_2 := concretizeAction(τ_k, PS);
                medt := append (buildParallelLts(concrete_1, concrete_2), medt);
            } else if (act := subtrace(t_i) ∧ act = τ_h ∧ t_i ∈ triple_i) then {
                concrete_1 := concretizeAction(τ_h, PS);
                medt := append(buildLts(concrete_1, medt));
            } else if(act := subtrace(t_j) ∧ act := τ_k ∧ t_j ∈ triple_i)then {
                concrete_2 := concretizeAction(τ_k, PS);
                medt := append(buildLts(concrete_2), medt);
            } else if (act := subtrace(t_i) ∧ t_i ∈ triple_i ∧ act = m̄_i) then {
                concrete_1 := concretizeAction(m_i, PS);
                concrete_2 := concretizeAction(m̄_i, PS);
                medt := append2(buildLts(RECEIVE, concrete_1), buildLts(SEND, concrete_2), medt);
            } else    // act := subtrace(t_i) ∧ t_i ∈ triple_i ∧ act = m_i
                concrete_1 := concretizeAction(m̄_i, PS);
                concrete_2 := concretizeAction(m_i, PS);
                medt := append2(buildLts(RECEIVE, concrete_2), buildLts(SEND, concrete_1), medt);
        }
        media := media ⋃ medt;
}
LTS mediator := transformToOneLTS(media);
mediator := makeDeterministic(mediator);
mediator := makeMinimal(mediator);
return mediator;
}
```

Listing 3.7: Mapping Phase Algorithm

3.6 Related Work

This chapter has concentrated on a theory to solve the protocol interoperability problem, which has been a key aspect in the research community for a long time. Many efforts have been done in several directions including for example formal approaches to protocol conversion [24, 49, 65], and their extension towards reducing the algorithmic complexity of protocol conversion [48]. A work strictly related to the theory presented in this chapter is [85] that proposes a *theory* to characterize and solve the interoperability problem of augmented interfaces of applications. The authors 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. Another perspective in the comparison between our work and [85] can be found in Section 4.5.

In recent years, a lot of work has been also devoted to behavioral adaptation in the Web Services research community, which has been actively studying this problem. Among these works, and related to our, there is [59]. 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 [75] where the theory underlying our approach is described at a high level and in [42] where a more detailed version of the theory can be found.

The work [30] addresses the interoperability problem between services and provide experimentation on real Web2.0 social applications. The paper deals with the integration of a new service implementation, to substitute a previous one with the same functionalities. The new implementation does not guarantee behavioral compatibility despite complying with the same API of the previous one. They hence 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.

3.7 Conclusion

In this chapter, we described our proposed theory towards 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. While the theory was introduced in the previous year deliverable D3.1 [5], this chapter has introduced a revised version that is based on a simpler abstraction and then matching for protocols.

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.

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 mediation (mediator synthesis). We are also concerned with the integration with complementary work ongoing within the CONNECT project so as to develop an overall CONNECT framework enabling the dynamic synthesis of emergent connectors among networked systems. Such an effort is more specifically reported in Deliverable D1.2 [3]. 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.

4 Abstract CONNECTor Synthesis

The previous chapter has recalled and further revised the CONNECT theory of mediators that was introduced in Year 1 and that sets the foundations of CONNECTor synthesis. As seen, CONNECTor synthesis relies on:

1. The abstraction of networked systems protocols in terms of LTS over the networked systems' observable actions whose semantics is given using ontologies;
2. The definition of a functional matching relation defined over networked systems' abstract protocols to identify whether protocols may coordinate using a mediator that solves possible behavioral mismatches;
3. The synthesis of a mediator that implements appropriate mapping between the protocols' actions and possible reordering of those actions; in other words, the mediators compose basic mediation patterns introduced in Chapter 2.

This chapter leverages the proposed theory, considering its practical application to real world networked systems. The notion of *mediator* underlying CONNECTors is not new. It has indeed been investigated since the need for interoperability in distributed systems was identified [49, 65]. However, this was initially a design-time concern, while today's dynamic distributed systems require on-the-fly adaptation. On-the-fly protocol adaptation has in particular been studied quite extensively in the context of Web services to deal with either dynamic service composition (e.g., [29]) or substitution (e.g., [26]). Still, existing work on runtime automated mediation concentrates on application-layer protocols, while the heterogeneity of open networked systems may concern both the application and middleware layers. Indeed, a one-size-fits-all middleware cannot be assumed. Middleware solutions will keep emerging according to specific requirements coming from application domains and/or networking environments.

Middleware interoperability solutions have been developed since the early days of middleware (see Deliverable D1.1 [2]). Indeed, one-to-one bridging was among the early approaches [66], which then evolved into more generic solutions such as Enterprise Service Bus [27], interoperability platforms [38] and transparent interoperability platforms [20, 60]. However, except for the transparent interoperability platforms, most of these solutions rely upon the design-time choice to develop applications using the proposed interoperability platform; and thus do not allow for on-the-fly interoperability between networked applications embedding different legacy middleware. Middleware interoperability further needs to cope with the many middleware interaction paradigms that now need to coexist. This includes accessing the same functionality through distinct paradigms (e.g., context-awareness through access to a data-centric sensor network or a RPC-based context server).

In light of the above, this chapter is concerned with:

1. The modeling of networked systems that is similar to the abstraction of networked systems protocols of our theory of mediators, while accounting for protocols that may build upon diverse middleware technologies, and may, in particular, rely on different interaction paradigms.
2. The refinement of the ontology mapping definition introduced in the theory, building upon the work on ontology, and further accounting for the semantics of actions regarding both the application and middleware layers.
3. The refinement of the functional matching relation and related mediator synthesis according to the proposed modeling of networked systems, where we in particular aim at reducing the complexity of protocol matching verification and mediator synthesis.

Section 4.1 introduces the proposed modeling of networked systems to enable reasoning about their functional compatibility/matching, while accounting for the heterogeneity of underlying middleware protocols. As further presented in Section 4.2, the modeling of networked

systems relies on ontologies that serve conceptualizing both middleware and application functions. Then, Section 4.3 introduces a refinement of the functional matching relation introduced in the previous chapter to lower the complexity of related reasoning, which is done at the expense of supported mediation patterns. This further leads to the direct synthesis of the mediator as presented in Section 4.4. Section 4.5 positions our work with respect to extensive effort in the area of protocol mediation where our contribution primarily lies in dealing with mediation from application down to the middleware layer. Finally, Section 4.6 concludes with a summary of the chapter's contribution and ongoing integration within the CONNECT architectural framework.

4.1 Modeling Networked Systems

As dicussed in the Description of Work of the CONNECT project and further elaborated in Deliverable D1.1 [2], the basic assumption we have for 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 [86]. Still, a crucial question 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 the matching of their respective functionalities. We call such functionalities *affordances* and we say that networked systems *match* when a networked system requires an affordance that matches an affordance provided by the other (Section 4.1.1). In the theory of mediators introduced in the previous chapter, affordances are inferred from the set of complementary coordination policies, given the specification of the networked systems' interaction behaviors as abstract protocols. However, to reduce the complexity of networked systems matching, we make explicit the specification of the networked systems' affordances, in a way similar to the specification of capabilities in the definition of semantic Web services. Then, the specification of networked systems decomposes into a number of affordances whose behaviors are defined as protocols over the networked system's observable actions. Observable actions are typically specified as part of the system's interface signature (Section 4.1.2) while the modeling of protocols relies on some concurrent language and may be advertised by the system or be possibly learned as investigated in WP4 (Section 4.1.3). Last but not least, the semantics of observable actions need to be rigorously defined in order to assign the same meaning to actions in any environment, for which we exploit ontologies.

4.1.1 Affordance

An *affordance* denotes a high-level functionality provided or required from the networked environment. Concretely, an affordance is specified as a tuple:

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

where:

- *Type* stands for required (noted *Req*), provided (noted *Prov*) or required and provided (noted *Req_Prov*) affordance. A provided affordance denotes an affordance offered in the network while a required one is to be consumed. A required and provided affordance is then both consumed and offered by the networked system, as common in peer-to-peer systems
- *Op* gives the semantics of the functionality associated to the affordance in terms of an ontology concept

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

- *I* (resp. *O*) specifies the set of inputs (resp. outputs) of the affordance, which is defined as a tuple $< i_1, ..., i_n >$ (resp. $< o_1, ..., o_m >$) with $i_{l=[1..n]}$ (resp. $o_{l=[1..m]}$) being an ontology concept

As an illustration, the *consumer* affordance of the Photo Sharing scenario is defined as:

Photo-Sharing_Consumer = <*Req, Photo-Sharing_Consumer,* <*Comment*>, <*Photo*>>,

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

4.1.2 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 tight to a specific middleware solution, while we target pervasive networking environments hosting heterogeneous middleware solutions. The specification of an action should then be enriched with the one of the middleware function that is specifically used to carry out that action; indeed, an observable action in an open pervasive network is the conjunction of an application-layer with a middleware-layer function. Middleware functions then need to be unambiguously characterized, which leads us to introduce a middleware ontology that defines key concepts associated with state-of-the-art middleware API, as presented in the next section. Given the above, the interface of a networked system is defined as a set of tuples. More formally:

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

where:

- m_f denotes a middleware function;
- a denotes the application action;
- I (resp. O) denotes the set of inputs (resp. outputs) of the action.

Moreover, as detailed in Section 4.2, the tuple elements are ontology concepts so that their semantics may be reasoned upon.

As an illustration, 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 Chapter 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.

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

[3] http://www.w3.org/TR/soap/

```
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 }
```

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

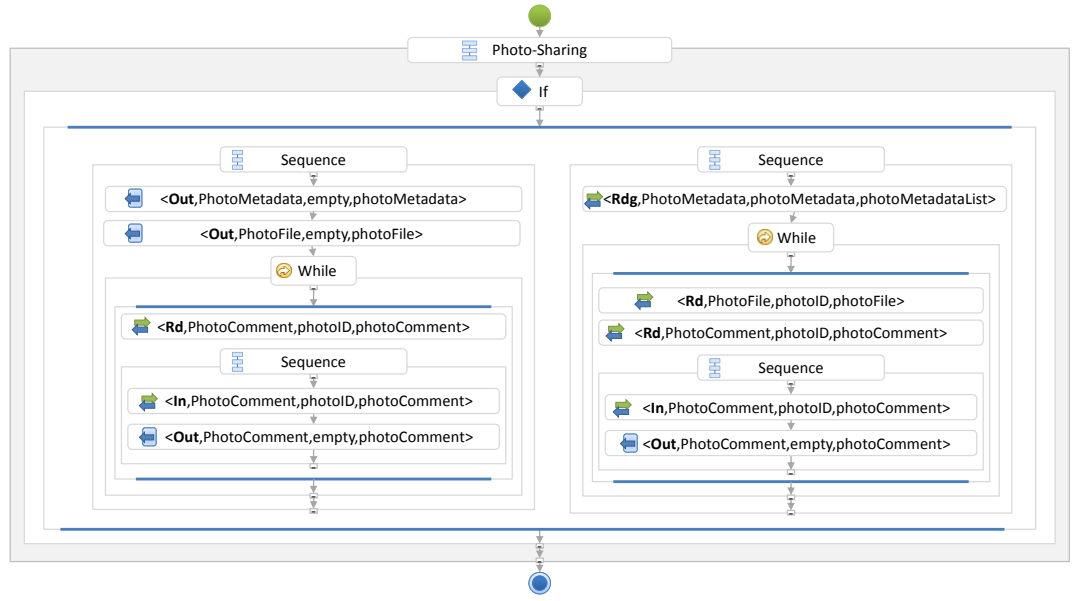
```
1  Interface_photo_sharing = {
2    <Out, PhotoMetadata, ∅, <photoMetadata>>,
3    <Out, PhotoFile, ∅, <photoFile>>,
4    <Rdg, PhotoMetadata, <photoMetadata>, <photoMetadataList>>,
5    <Rd, PhotoFile, <photoID>, <photoFile>>,
6    <Rd, PhotoComment, <photoID>, <photoComment>>,
7    <Out, PhotoComment,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

4.1.3 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, 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 as investigated in WP4 but this is beyond the scope of this deliverable. 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.

In our work, we more specifically 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

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



a) LIME-based Implementation



b) SOAP-based Implementation

Figure 4.1: Infrastructure- and peer-to-peer-based Photo Sharing

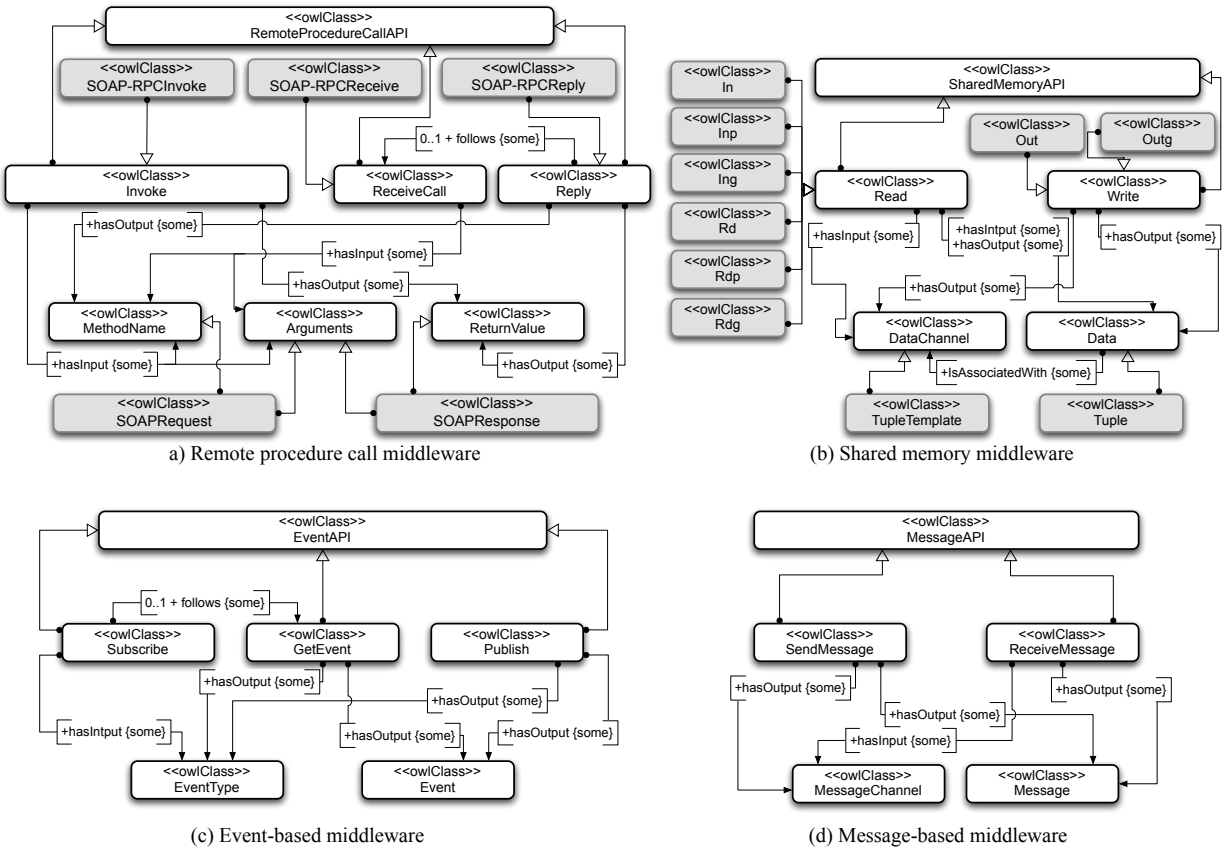


Figure 4.2: Middleware ontology

independent actions; (ii) the formal specification of BPEL in terms of process algebra that allows abstracting BPEL processes for automated reasoning about protocol matching [36]; and (iii) the rich tool sets coming along with BPEL, which in particular ease process definition 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.

For illustration, Figure 4.1 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.

4.2 Ontology for Mediation

Realizing CONNECTors primarily relies on reasoning about affordance matching together with identifying matching observable actions among the actions performed by networked systems.

As discussed in the previous chapter, ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of pervasive networked systems. Indeed, "*an ontology is a formal, explicit specification of a shared conceptualization*" [78]. 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 [35].

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 [11].

Towards enabling CONNECTors, we introduce a middleware ontology that forms the basis of middleware protocol mediation (Section 4.2.1). In addition, domain-specific application ontologies characterizing application actions serve defining both control- and data-centric concepts (Section 4.2.2).

4.2.1 Middleware Ontology

State-of-the-art middleware may be categorized according to four *middleware types* regarding provided communication and coordination services [79]: *remote procedure call, shared memory, event-based* and *message-based*. As depicted in Figure 4.2 and more specifically with concepts defined in white boxes, the proposed middleware ontology is structured around these four categories, which serve as reference enabling to align functions of different middleware solutions. Indeed, the reference middleware ontology can be refined into concepts associated with functions of a specific middleware. This is illustrated in the figure by the grayed boxes that define concepts of the LIME and SOAP-based middleware solutions that we specifically consider in our Photo Sharing scenario. In addition to the *is-a* relation that is denoted by a white arrow, the middleware ontology introduces a number of customized relations between concepts: *hasOutput* (resp. *hasInput*) to characterize output (resp. input) parameters. We also use relations from best practices in ontology design[6] as illustrated by the *follows* relation that serves defining sequence patterns.

The ontology is given as a set of UML diagrams. In Figure 4.2.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 4.2.b). In the latter, messages are exchanged using the *SendMessage* and *ReceiveMessage* functions parameterized by the actual message and related channel (see Figure 4.2.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 4.2.c).

The proposed ontology serves aligning the functions of middleware of the same middleware type through mapping onto the reference functions, which is illustrated for the specific cases of SOAP-based and LIME middleware. Heterogeneity in the underlying implementation may then be overcome using transparent middleware interoperability solutions (e.g., [21]).

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

[6] http://ontologydesignpatterns.org

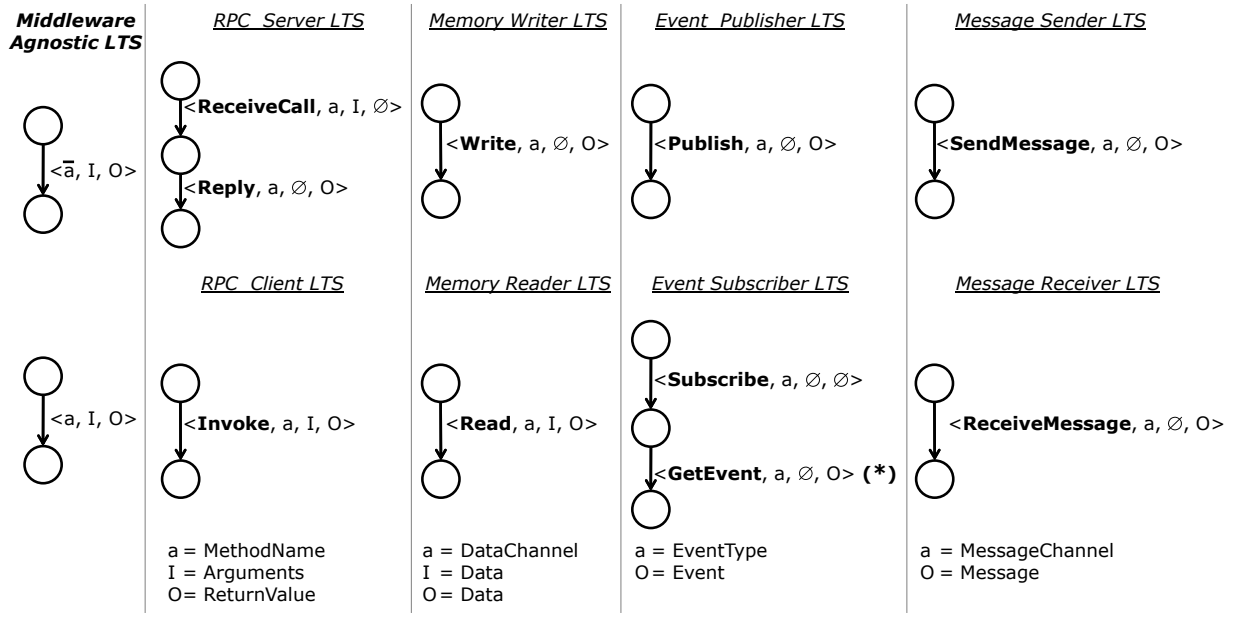


Figure 4.3: Middleware alignment

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



Figure 4.4: Shared-memory based Photo Sharing

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



Figure 4.5: Middleware-agnostic peer-to-peer Photo Sharing

The given alignments abstract protocols associated with the realization of affordances as *middleware-agnostic processes*. 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 from the reference ontology, which are then translated into input and output actions through the defined alignment. This is illustrated in Figures 4.4 and 4.5, which depict the protocol associated with the peer-to-peer Photo Sharing implementation, after abstracting middleware-specific functions into reference functions (Figure 4.4) and further aligning onto middleware-agnostic input and output application-specific actions (Figure 4.5). 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.

4.2.2 Application-specific Ontology

The *subsumption* relation of ontologies serves matching *application-specific actions* against each other. Basically, and 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.

For illustration, Figure 4.6 gives an excerpt of the domain-specific ontology associated with our Photo Sharing scenario, which shows the subsumptions holding among the various concepts defining the interfaces of the networked systems implementing the scenario.

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

Finally, subsumption is not the panacea to reason about semantic relationships between concepts and many other relations such as sequence [32] 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.

[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

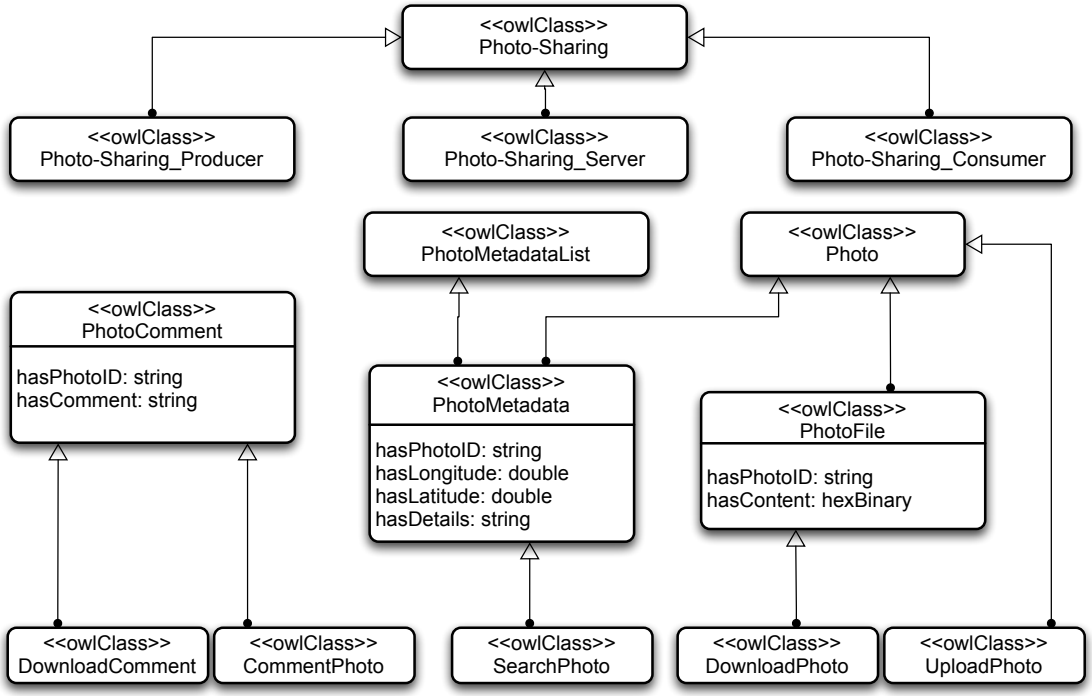


Figure 4.6: Photo Sharing ontology

4.3 Functional Matching of Networked Systems

Given the models characterizing networked systems that are introduced in Section 4.1 and related ontology definition, CONNECTors are enabled through matching and mapping functions defined over the actions of networked systems, whose definition builds on the theory of mediators of the previous chapter. Precisely, if two networked systems implement *semantically matching affordances* (Section 4.3.1), then we check whether the affordances indeed *behaviorally match* (Section 4.3.3) under the *mapping of their interfaces* (Section 4.3.2) according to their ontology-based semantics. Compared to the theory of mediators, the semantic matching of affordances is introduced so as to limit the use of reasoning about behavioral matching, and hence improve the overall performance of the abstract mediator synthesis.

4.3.1 Semantic Matching of Affordances

The first step in identifying the possible *compatibility* of two networked systems is to assess whether they respectively provide and require a *semantically matching affordances*. More precisely, and following the definition of [69], we say that affordance $Aff_R = <$ *Req, Op*$_R$, *I*$_R$, *O*$_R$ $>$ semantically matches with affordance $Aff_P = <$*Prov, Op*$_P$, *I*$_P$, *O*$_P$ $>$, noted $Aff_R \hookrightarrow Aff_P$, *iff* in the given ontology:

- *Op*$_R \sqsubseteq$ Op$_P$,
- I$_P \sqsubseteq$ I$_R$ which is a shorthand notation for subsumption between sets of ontology concepts, and
- O$_R \sqsubseteq$ 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 addition, since the affordance is related to semantic concepts, we make a similar assumption to the one made in the Semantic Web [69], i.e., by specifying Op_P as the functional concept, the provider commits to offering all the functionalities subsumed by Op_P and output consistent with every concept subsumed by O_P. If this is not the case, then the functionality/output should be restricted to those verifying the above assumption. Similarly, the requester commits to provide any input consistent with the classes that I_R subsumes. However, if the input/output are related to syntactic (XML-based) types and not to semantic concepts, it becomes important to verify the Liskov Substitution Principle (LSP) [51] in the following way:

- Op_P `subtypeOf` Op_R, which corresponds to the LSP co-variance rule.
- I_R `subtypeOf` I_P, which corresponds to the LSP contra-variance rule for the outputs
- O_P `subtypeOf` O_R, , which corresponds to the LSP co-variance rule.

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

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

$$PhotoSharingConsumer = <\textit{Req, Photo-Sharing_Consumer,} <\textit{PhotoComment}>, <\textit{Photo}>> \quad (1)$$

$$PhotoSharing = <\textit{Req_Prov, Photo-Sharing,} <\textit{Photo}> \vee <\textit{PhotoComment}>, <\textit{Photo, PhotoComment}>> \quad (2)$$

$$PhotoSharingConsumer \hookrightarrow PhotoSharing \quad (3)$$

Finally, in the case where both affordances are both provided and required, the equivalence of concepts obviously needs to hold.

Given the semantic matching of affordances, the CONNECTor between the corresponding matching networked systems should mediate –if possible– behavioral 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 behavior 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 [24] and thus requires heuristics to make the problem tractable. Towards that goal, we account for the basic mediation patterns described in Chapter 2 as follows:

- **Messages ordering pattern** (manages ordering mismatch): Concerns the re-ordering of actions so that networked systems may indeed coordinate. Assuming the specification of affordance behavior using a BPEL-like language as discussed in Section 4.1.3, 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** (handles extra output action or missing input action mismatch): As discussed above, extra output actions are simply discarded from the standpoint of behavioral matching. Obviously, the associated mediator should handle any extra synchronous output action to avoid deadlock.
- **Message producer pattern** (addresses missing output action or extra input action 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** (manages mismatches related to splitting of actions): Splitting actions relate to having an action of one system realized by a number of actions of the other. Then, an input action may be split into a number of output actions of the matching networked system if such a relation holds from the domain-specific ontology giving the semantics of actions. On the other hand, we do not deal with the splitting of output actions, which is an area for future work given the complexity it introduces.

- **Message merger pattern** (handles mismatches related to merging of actions): The merging of actions is the dual of splitting from the standpoint of the matching networked system. Then, according to the above, we only handle the merging of output actions at this stage.

- **Message translator pattern** (solves signature mismatch): Concerns the translation of actions according to their semantic matching, as defined by the related middleware- and domain-specific ontologies.

4.3.2 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 towards the realization of a given affordance. Considering the definition of the theory of mediators of previous chapter, interface mapping corresponds to the *ontology mapping* of the theory, provided the above-mentioned restrictions regarding the supported mediator patterns.

Let two networked systems that respectively implement the semantically matching affordances Aff_1 and Aff_2. Let further $\mathcal{I}_{Aff_1}$ (resp. $\mathcal{I}_{Aff_2}$) be the set of middleware-agnostic actions executed by the protocol realizing Aff_1 (resp. Aff_2); $\mathcal{I}_{Aff_1}$ and $\mathcal{I}_{Aff_2}$ are then subsets of the actions defined in the networked systems' interfaces, which are further made middleware-agnostic according to the alignment defined in Section 4.2.1. We introduce the function $Map_I(\mathcal{I}_{Aff_1}, \mathcal{I}_{Aff_2})$ that 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. Formally:

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

where:

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

and

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

where $\prec$ denotes the inclusion of sequences. In the above definition, the ordering of actions given by the sequence follows from the sequencing of actions in the protocol realizing the 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 below the interface mapping between the actions of *PhotoSharingConsumer* and those of *PhotoSharing*. All the input actions of *PhotoSharingConsumer* have a corresponding output action in *PhotoSharing*. On the other hand, the input actions of *PhotoSharing* associated with the production of photos do not have matching output actions in *PhotoSharingConsumer*. As a result, we support the adaptation of protocols for interaction between *PhotoSharingConsumer* and *PhotoSharing* regarding the consumption of photos by the former only, as further discussed in the next section.

$Map(\mathcal{I}'_{photo_sharing_consumer}, \mathcal{I}'_{photo_sharing})$= {
$< SearchPhotos, < photoMetadata >, < photoMetadataList >>$
$\mapsto \{<< \overline{PhotoMetadata}, \emptyset, < photoMetadata >>>\},$
$< DownloadPhoto, < photoID >, < photoFile >>$
$\mapsto \{<< \overline{PhotoFile}, \emptyset, < photoFile >>>\},$
$< CommentPhoto, < photoComment >, < acknowledgment >>$
$\mapsto \{<< \overline{PhotoComment}, \emptyset, < photoComment >>>\},$
$< DownloadComment, < photoID >, < photoComment >>$
$\mapsto \{<< \overline{PhotoComment}, \emptyset, < photoComment >>>\},$
$< PhotoComment, < photoID >, < photoComment >> \mapsto \emptyset,$
$< PhotoMetadata, < photoMetadata >, < photoMetadataList >> \mapsto \emptyset,$
$< PhotoFile, < photoID >, < photoFile >> \mapsto \emptyset$
}

We are currently devising an algorithm towards efficiently computing interface mappings, building upon related effort in the area (e.g., [59]). In addition, as part of the development of the Discovery enabler within WP1 (see Deliverable D1.2 [3]), 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 discussed next, using ontology-based model checking.

4.3.3 Behavioral Matching of Affordances

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

Let $\mathcal{I}'_1 = \{\alpha_i = \langle a_i, I_{a_i}, O_{a_i}\rangle\}_{i=1..n} \cup \left\{\overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j}\right\rangle\right\}_{j=1..m}$ be the abstract interface associated with required affordance Aff_1, and

$\mathcal{I}'_2 = \{\alpha'_{i'} = \langle a'_{i'}, I_{a_{i'}}, O_{a_{i'}}\rangle\}_{i'=1..n'} \cup \left\{\overline{\beta'_{j'}} = \left\langle \overline{b'_{j'}}, I_{b'_{j'}}, O_{b'_{j'}}\right\rangle\right\}_{j'=1..m'}$ be the abstract interface associated with provided affordance Aff_2.

From $Map_I(\mathcal{I}'_1, \mathcal{I}'_2)$, we have: $\forall \alpha_{i=1..n} \in \mathcal{I}'_1 : \alpha_i \mapsto \left\langle \overline{\beta'_1}, ..., \overline{\beta'_n}\right\rangle \mid \beta'_j \in \mathcal{I}'_2$

We then define the processes $M_{\alpha_{i=1..n}}$ that deal with the splitting of actions by allowing the synchronization of each input action $\alpha_{i=1..n}$ with its corresponding output actions:

$$M_{\alpha_{i=1..n}} \stackrel{\text{def}}{=} \beta'_1 \rightarrow ... \rightarrow \beta'_n \rightarrow \overline{\alpha_i} \rightarrow M_{\alpha_{i=1..n}}$$

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

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

We define similarly $M_{\alpha'_{i'=1..n'}}$ and $M_{\beta_{j=1..k}}$ for Aff_2.

A process P_1 associated with affordance Aff_1 *behaviorally matches* a process P_2 associated with affordance Aff_2 under $Map(\mathcal{I}'_1, \mathcal{I}'_2)$, noted $P_1 \hookrightarrow_{\mathcal{P}} P_2$, iff

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

where $\leq$ refers to trace refinement as defined in [72] and guarantees that *mediated* P_1 can safely communicate with *mediated* P_2. Reasoning about behavioral matching may then be implemented using model checking [44], supporting efficient online reasoning is part of our ongoing work.

Applying the above definition to our Photo Sharing example, we can check that:

$$P_{photo_sharing_consumer} \hookrightarrow_{\mathcal{P}} P_{photo_sharing}$$

4.4 Mediator Synthesis

Following all the above provided definitions, the mediator behavior is direct and is defined as:

$$Mediator = \left(\underset{i=1..n}{||} M_{\alpha_{i=1..n}} \right) || \left(\underset{j'=1..k'}{||} M_{\beta'_{j'=1..k'}} \right) || \left(\underset{i'=1..n'}{||} M_{\alpha'_{i'=1..n'}} \right) || \left(\underset{j=1..k}{||} M_{\beta_{j=1..k}} \right)$$

From a pragmatic standpoint, the synthesis of mediators further requires special care 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 ontology 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. This is the focus of the next chapter that deal with concrete CONNECTor deployment.

4.5 Related Work

Protocol interoperability has been the focus of significant research since the early days of networking. This has initially led to the study of systematic approaches to protocol conversion (i.e., synthesizing a mediator that adapts the two interacting protocols that need to interoperate) based on formal methods as surveyed in [24]. 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 [49] or correspond to the reference protocol associated with the service to be realized through protocol interaction [65]. 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 [24]. 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 [48]. Our work extensively builds on these formal foundations, adopting a bottom-up approach in the form of interface mapping. However, unlike the work of [49], our interface mapping is systematically inferred, thanks to the use of ontologies. In addition, while the proposed formal approaches pave the way for rigorous reasoning about protocol compatibility and conversion, they are mostly theoretical, dealing with simple messages (e.g., absence of parameters).

More practical treatment of protocol conversion is addressed in [85], which focuses on the adaptation of component protocols for object-oriented systems. The solution is top-down in that the synthesis of the mediator requires the mapping of messages to be given. By further concentrating on practical application, the authors have primarily targeted an efficient algorithm for protocol conversion, leading to a number of constraining assumptions such as synchronous communication. In general, the approach is quite restrictive in the mediation patterns that it supports by not buffering messages and thus preventing the handling of the merging/splitting or re-ordering of messages in general. Then, while our solution relates to this specific proposal, it is more general by dealing with more complex mediation patterns and further inferring message mapping from the ontology-based specification of interfaces. Our solution further defines protocol compatibility by in particular requiring that any input action (message reception) has a

corresponding (set of) output action(s), while the definition of [85] requires the reverse. Our approach then enforces networked systems to coordinate so as to update their states as needed, based on input from the environment.

More recently, with the emergence of Web services and advocated universal interoperability, the research community has been studying solutions to the automatic mediation of business processes [81, 80, 58, 84]. 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, 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 manipulates formally grounded process models.

Moreover, within the Web services area, the research 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 [58, 71], latest work concentrates on automated, run-time solutions [26]. Our work closely relates to the latest effort, sharing the exploitation of ontology to reason about interface mapping and the further synthesis of protocol converter behaviors according to such mapping, using model checking [26]. However, our work goes one step further by not being tight to the specific Web service domain but instead considering highly heterogeneous pervasive environments where networked systems may build upon diverse middleware technologies and hence protocols.

Our work also closely relates to significant effort from the semantic Web service domain and in particular the WSMO (Web Service Modeling Ontology) initiative that defines mediation as a first class entity for Web service modeling towards supporting service composition. The resulting Web service mediation architecture highlights the various mediations levels that are required for systems to interoperate in a highly open network [76]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [29, 81]. However, as for the above-mentioned work on Web service adaptation, mediation is focused on the upper application layer, ignoring possible mismatches in the lower protocol layers. In other words, work from the Web service arena so far concentrates on interoperability among networked systems from the same technology domain. However, pervasive networks will increasingly be populated by highly heterogeneous systems, spanning, e.g., from systems for sensing/actuating to enterprise information systems. As a result, systems run disparate middleware protocols that need to be reconciled on the fly.

The issue of middleware interoperability has deserved a great deal of attention since the emergence of middleware. Solutions were initially dealing with diverging implementations of the same middleware specification and then evolved to address interoperability among different middleware solutions, acknowledging the diversity of systems populating the increasingly complex distributed systems of systems. As reported in Deliverable D1.1 [2] and further formalized in Deliverable D3.1 [5], one-to-one bridging was among the early approaches [66] and then evolved into more generic solutions such as Enterprise Service Bus [27], interoperability platforms [38] and transparent interoperability platforms [20, 60]. Our work takes inspiration from the latest transparent interoperability approach, which is itself based on early protocol conversion approaches. Indeed, protocol conversion appears the most flexible approach as it does not constrain the behavior of networked systems. Then, our overall contribution comes from the comprehensive handling of protocol conversion, from the application down to the middleware layers, which have so far been tackled in isolation. In addition, existing work on middleware-layer protocol conversion focuses on interoperability between middleware solutions implementing the same interaction paradigm. On the other hand, our approach allows for interoperability among networked systems based upon heterogeneous middleware paradigms, which is crucial for the

increasingly heterogeneous pervasive networking environment.

4.6 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 CONNECT*ors* , which adapt the protocols run by networked systems that functionnally match but possibly mismatch from the standpoint of their interaction protocols and even middleware technology used for interactions. Enabling CONNECTors specifically lies in the appropriate modeling of the networked systems' high-level functionalities and related protocols, for which we exploit ontologies so as to enable unambiguous specification. Compared to related work that deals with either automated protocol conversion/mediation or middleware interoperability, our contributions lies in comprehensively dealing with both the application and middleware layers. In addition, through the alignment of middleware concepts, we are able to deal with interoperability between networked systems relying on heterogeneous middleware paradigms.

While this chapter has surveyed the overall model-based approach enabling CONNECTors, it comes along with concrete enablers to be deployed in the network for actual enactment of the CONNECTors [15], as studied in WP1. Enablers include *universal discovery*, which in particular implements the matching and mapping relations discussed in this chapter, so as to enable networked systems to meet and compose on the fly. However, it should be acknowledged that most legacy systems do not advertise interfaces like the ones needed by CONNECTors but instead advertise simple interface signatures, as common with today's middleware. This leads the CONNECT project to investigate *learning enablers* so as to enable automated learning of interaction protocols [17, 40] as well as inference of affordances from interface signatures. Furthermore, while universal discovery enables networked systems to compose abstractly through the proposed abstract CONNECTor synthesis, concrete CONNECTors need to be instantiated. Concretization of CONNECTors is the focus of the next chapter.

5 From Abstract to Concrete Mediator Synthesis and Deployment

The construction of the behavioral model of the needed CONNECTor remains the initial step of the synthesis process. An effective connection of several networked systems also requires the translation of this model into an executable artifact that can be deployed and run. This chapter specifically addresses this final step, and goes from the construction to the deployment and the starting up of the final CONNECTor artifact.

The remainder of this chapter is divided as follows. Section 5.1 gives a brief overview of the runtime architecture of CONNECTors and discusses two possibles solutions envisioned to construct them: generation of code and interpretation of the CONNECTor model. Section 5.2 presents the model we used to described CONNECTors and how it can be derived from the LTS models outputted earlier during the synthesis process. Section 5.3 explains how to construct a run-time CONNECTor by generating the code that implements the related CONNECTor model. Section 5.4 briefly surveys an alternative solution based on the direct interpretation of CONNECTor models that will be further investigated during the third year of the project. Finally, Section 5.5 illustrates the construction of the run-time CONNECTor using the Photo Sharing example introduced in Section 1.5.

5.1 Overview

This section overviews the construction of running CONNECTors. The first part introduces the general run-time architecture of CONNECTors and explains how they combine an *ad hoc* element dynamically created with several *interoperability proxies* that are reused from one CONNECTor to another. The second part focuses on these proxies and explains their interaction with the core of the CONNECTor. Finally, the last part compares the two approaches envisioned to implement the core of the CONNECTor: the generation of code and the dynamic interpretation of the CONNECTor model. These two approaches are later detailed in Section 5.3 and Section 5.4, respectively.

5.1.1 Run-Time Architecture of a CONNECTor

As shown in previous deliverables [2], the main obstacles to *eternally connected systems* are interoperability issues that may arise at the application level, at the middleware level or because of some mismatches on data semantics. Intuitively, the synthesis process results in a new CONNECTor that addresses each of these issues.

However, from the architectural perspective, some issues, such as the encoding of network messages, do not depend on the application-level, and the technical solutions to these issues can therefore be reused from one CONNECTor to another. The resulting runtime architecture thus combines *generic* elements addressing application-independent issues with an *ad hoc* element, so-called *mediator*, addressing the application-dependent issues. Figure 5.1 below illustrates the runtime architecture of a CONNECTor enabling communications between two applications deployed on different middleware technologies.

While acting as a bridge between the two applications, the CONNECTor must be able to properly communicate with both sides regardless of the middleware technologies at play in the collaboration. The part of the CONNECTor in charge of the communications with a given middleware (so-called *Proxy*) is not actually synthesized from the CONNECTor model, but reused and configured with the relevant descriptions of the middleware technologies. Only the mediator is generated from the CONNECTor model. The description of the interoperability proxies shall therefore not be further detailed in this deliverable, but can be found in deliverable D1.2.

These *proxies* components provide two facilities. On one hand, they read incoming messages on the network and build relevant ”abstract messages” that can be understood by the

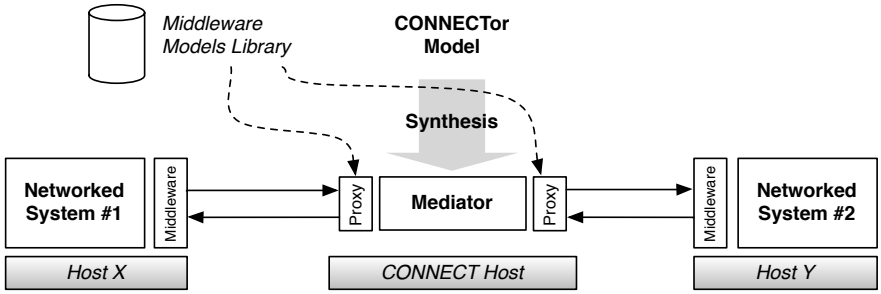


Figure 5.1: Run-time Architecture of CONNECTors

mediator. On the other hand, they compose proper middleware messages given ”abstract messages” provided by the mediator.

The last element, which is not represented on Figure 5.1 for the sake of clarity, is the interaction between the running CONNECTor and the monitoring framework provided by the WP5. The monitoring framework must indeed observe the behavior of the running CONNECTor in order to detect some potential failures. This is further detailed in Section 5.3.4, which deals with the deployment of *ad hoc* CONNECTors.

5.1.2 Interaction with the Interoperability Proxies

As mentioned above, the proxies are in charge of decrypting and composing middleware-specific messages. Proxies and mediator indeed exchange *abstract messages* that are independent from any specific middleware.

The communications between the mediator and the interoperability proxies can be synchronous or asynchronous. Using *synchronous* communications (c.f. Figure 5.2), the interoperability proxies call the mediator to notify that a new network message has been received. This is actually a blocking call, which prevents the interoperability proxy to receive any other messages occurring while the mediator is processing the message. By contrast, using asynchronous communications (c.f. Figure 5.3), the interoperability proxy and the mediator are two active entities (e.g., implemented as separate threads in Java), which produce and consume abstract messages, respectively. The interoperability proxy produces messages in a shared pool, whereas the mediator extracts and processes these messages. Even if the mediator is currently processing a message, the interoperability framework can still add new messages into the pool: the mediator will process them later.

Whatever communication scheme is applied, all the complexity of the mediator implementation remains in the processing of an event, which must be done with respect to both the event and the current state of the mediator (as defined in the CONNECTor model outputted by the synthesis).

5.1.3 Code Generation vs. Model Interpretation

There are basically two possible ways to envision the realization of runtime CONNECTors from software models, namely: the generation of executable code (C, Java, etc.) or the dynamic interpretation of these CONNECTor models. These two approaches both assume the existence of *interoperability proxies* that ensure proper communications with versatile middleware technologies. The realization process is therefore boiled down to the implementation of the CONNECTor logics and does not deal with any low-level communication issues.

On the one hand, the code generation approach (Figure 5.4) suggests to first translate the CONNECTor model into an executable artifact ensuring the mediation between the networked systems at play. The CONNECTor is then linked to the interoperability proxies at run-time, as-

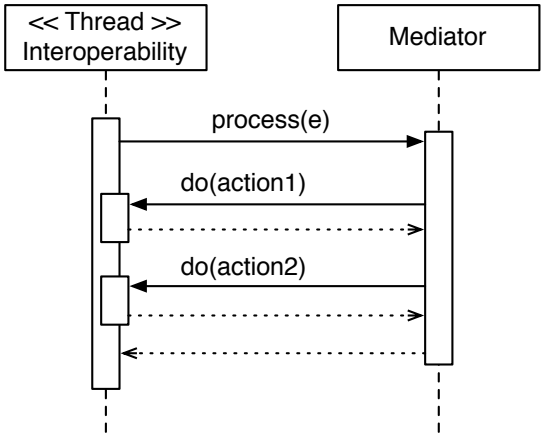


Figure 5.2: Synchronous communication pattern between the CONNECTor Core and the Interoperability Proxies

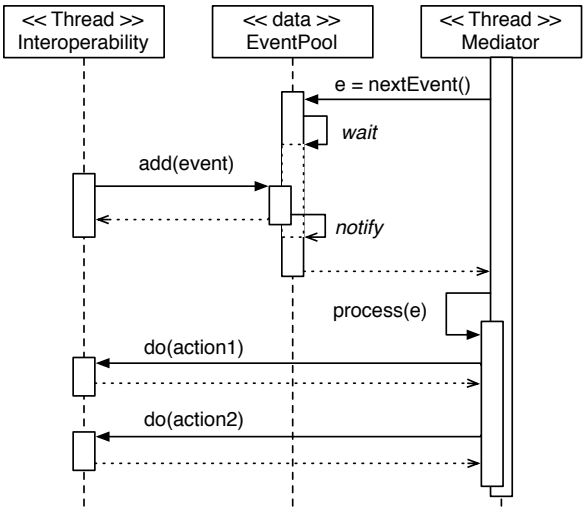


Figure 5.3: Asynchronous communication pattern between the CONNECTor Core and the Interoperability Proxies

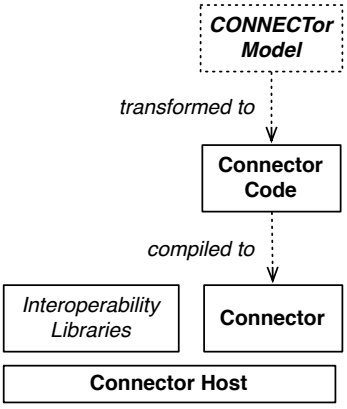


Figure 5.4: Compilation of CONNECTor models

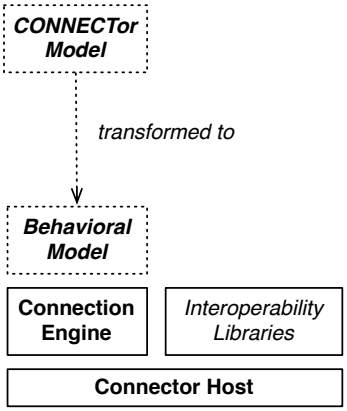


Figure 5.5: Runtime interpretation of CONNECTor models

suming that these proxies are already deployed on the target host. The main benefits of this approach is the efficiency of the resulting CONNECTor, where all possible mediation actions are explicitly hardcoded using low level mechanisms such as conditional branching or dynamic binding. The main drawback is the low degree of flexibility and maintainability, which calls for a re-generation in case of failure or ill-behavior at runtime.

On the other hand, the interpretation approach (Figure 5.5) advocates the dynamic mediation between the peers at play with respect to a given CONNECTor model. The interpretation engine, which appears here, dynamically computes the mediation actions that must be performed using the embedded CONNECTor model, and executes them using the interoperability proxies, deployed on the targeted host. By contrast with the generative approach, the use of an interpretation engine increases the flexibility and maintainability of runtime CONNECTors, but decreases the overall efficiency due to the need of additional online computation.

These two approaches are more complementary rather than conflicting. A generative approach better suits perennial communications, between well-known legacy systems whose characteristics are not evolving anymore. Interpretation would better fit transient or situational communications, where networked systems are likely to appear or disappear dynamically. The two following sections detail both approaches, respectively.

5.2 Modeling Run-Time CONNECTors

The meta-model that is presented in Figure 5.6 formalizes the language of CONNECTors. Models conforming to this meta-model can be used as an input of the code generator as we shall see in Section 5.3 or can be dynamically interpreted as we shall discuss in Section 5.4. The basic idea is to model a runtime CONNECTor as a mealy machine, extended with a memory. This memory is used to store and propagate values between incoming and outgoing messages. With respect to the terminology of mealy machine, we describe the CONNECTor model in terms of events (reception of an incoming message) and action (emission of an outgoing message). Another difference with standard mealy machines is that events and actions are prefixed by the partner concerned: similar messages received from different partners must trigger different transitions, and similar messages may be potentially sent to different partners.

This syntactic model is further restricted by the following OCL constraints. The constraint below ensures that each event specified as a transition trigger is actually declared in the the context of the CONNECTor.

```
context Connector :
  inv: transitions.forall{ t | partners.include(t.trigger.source)}
```

Besides, there are also additional constraints ensuring that transitions do not branch between states that does not belong to the same automaton. This is specified by the following four constraints over states and transitions.

```
context Connector:
  inv: transitions.forall{ t | states.include(t.source)}
  inv: transitions.forall{ t | states.include(t.target)}
  inv: states.forall{ s | s.outgoing.forall{ t | transitions.include(t)}}
  inv: states.forall{ s | s.incoming.forall{ t | transitions.include(t)}}
```

5.3 Generation of *ad hoc* CONNECTors

This section describes how we construct an *ad hoc* running CONNECTor from a given CONNECTor model. This initial solution is based on the direct generation of imperative code, and this section is consequently organized following the basic steps of the process: code generation, compilation and packaging, and deployment and starting up.

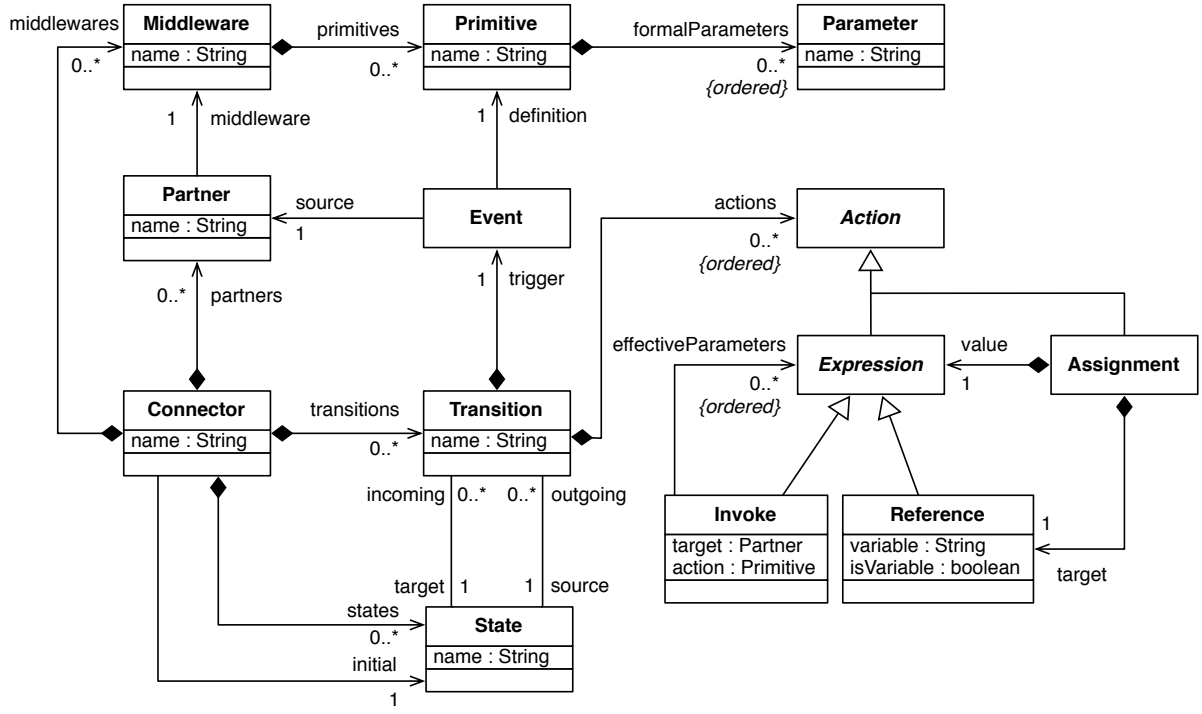


Figure 5.6: Syntactic Domain of Run-time CONNECTors

The first part of this section briefly surveys the nine main strategies that can be used to implement an LTS-like model using an *imperative* language such as C or Java. Then, we illustrate how to use the simplest one, namely the use of two *nested switch statements*, to generate the code realizing the core part of the CONNECTor, namely the mediator. The two following parts address the compilation and the deployment issues, respectively. A brief outline of the architecture of the associated prototype concludes this description of the generation of *ad hoc* CONNECTors.

5.3.1 Existing Code Generation Strategies for LTS Models

From the implementation perspective, a mealy machine can be seen as a function specifying which actions must be undergone with respect to the current state and with respect to the event that occurs. More formally, the function m encoded by a mealy machine can be described as: $m : S \times E \rightarrow S \times A$ where S is the set of possible states, E is the set of possible events, and A the set of possible actions.

At the code level, the possible actions are encoded into blocks of code performing the action, which may or not be encapsulated into separated procedures. The implementation of the function encoded into the machine is therefore boiled down to a mechanism that correctly switches to the relevant block of code with respect to the current state of the machine and the event to process.

As far as we know, existing programming paradigms have resulted into three mechanisms that can be used to perform conditional branching, namely: the explicit conditional branching of imperative languages, additional high-order programming facilities and the dynamic binding provided by object-oriented (OO) languages.

Explicit Conditional Branching (ECB) is the ability to jump to a different part of a program based on a certain condition being met. In most imperative languages, a statement such as `if` c `then` b_1 `else` b_2 specifies that the the block b_1 will be executed if and only if the condition c is evaluated as true and that the block b_2 will be executed otherwise.

High-Order Programming (HOP) is the ability to use functions as values; it is borrowed from computation models like lambda calculus, which make heavy use of higher-order functions. For instance, the function $\lambda xy.x(y)$ is a high-order function which applies the function x received as a first input on the value y received as a second input. High-order programming provides another solution to conditional branching by enabling the storage and execution of code blocks encapsulated into functions.

Dynamic Binding (DB) is the ability of object-oriented languages (OO) to determine the exact implementation of a request based on both the request (operation) name and the receiving object at run-time. For instance, a call such as $o_1.m_1()$ depends on the actual class of the object o_1, which is potentially unknown at design time. Dynamic binding provides another solution to conditional branching by enabling switching between code blocks encapsulated into methods of different classes.

Table 5.1: Overview of the possible implementation patterns, with respect to the existing branching mechanisms

Branching Mechanism		**Pattern Description**
State	*Event*	
ECB	ECB	Nested Switch Statements [82]
	HOP	Not Documented
	DB	Command Pattern [37]
HOP	ECB	Not Documented
	HOP	Transition Table Pattern [25, 31]
	DB	Command Pattern [37] (alternative)
DB	ECB	State Pattern [37]
	HOP	State Action Mapper [68]
	DB	State & Command Patterns [28]

The function encoded by a mealy machine depends on both the current state of the machine and the event to process. Therefore, its implementation requires two branching mechanisms: one branching with respect to the current state and the other one branching with respect to the event to process. The three ”branching” mechanisms introduced above lead to the nine possible implementation strategies of the mealy machines summarized by Table 5.1.

Among these possible implementation patterns, some have already been well-documented such as the Command and State patterns [37]. In addition, some hybrid solutions logically appear but do not necessarily provide significant benefits.

From the perspective of CONNECT , the selection of the most relevant pattern mainly depends on the degree of flexibility that we expect from the run time CONNECTor. Recovery actions that would manipulate the structure of the mealy machine are made possible by the use of object-oriented patterns, which reify different states by different objects. As the set of recovery actions

is still under investigation in WP5, the initial experiments reported in the rest of this chapter concern the simplest pattern presented, namely the use of nested switch statements.

5.3.2 Example: Using Nested Switch Statements

The main idea of the *nested switch* pattern is to use variables to reify both the current state of the mediator and incoming events. The transition function is then implemented using two nested conditional statements: the first one switching with respect to the current state, and the second one switching with respect to the current event.

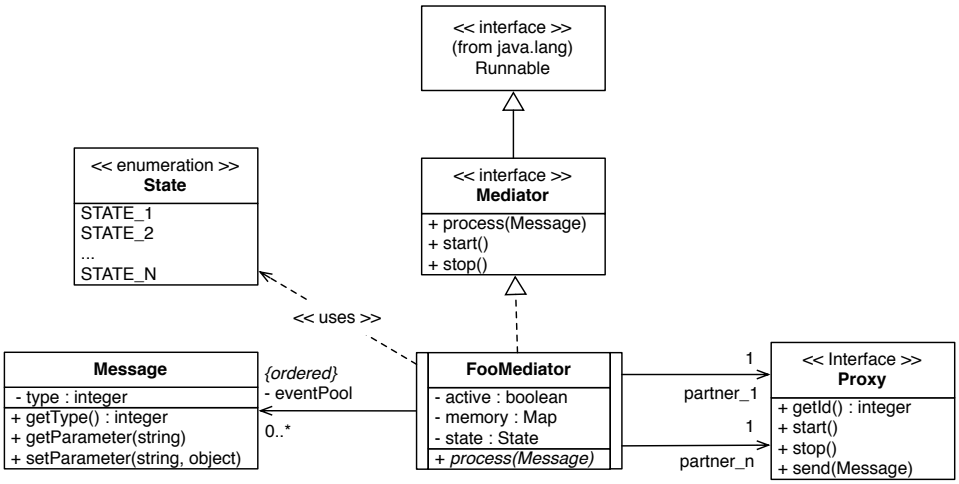


Figure 5.7: Detailed design of a mediator implemented using the nested switch pattern

Figure 5.7 illustrates with a UML class diagram, the different parts of the detailed implementation of a mediator using this pattern:

- The possible states of the mediator (i.e, of the initial mealy machine) are reified as an enumeration. In the related code sample below, this enumeration is declared on line 3. The current state of the mediator is stored in a specific attribute declared on line 12.
- Each partner involved in the collaboration is reified as an object, to which the mediator is linked. In the code below, the related proxy objects are declared as private attributes on line 7.
- The mediator object is equipped with a pool of messages where it can store the incoming messages that are received when the operation *process* is invoked. In the code below, the related attribute, named *eventPool*, is declared on line 13 and later used to store new messages on line 76.
- The mediator has a local memory, modeled as a hash map, in which it can store objects that are carried by incoming messages. This memory is declared on line 14 in the code below.

```
1 public class FooMediator implements Mediator {
2
3   private enum InternalState {
4     STATE1, STATE2,
5   }
6
7   private Proxy partner1;
8   private Proxy partner2;
9
```

```
10  private Thread internal;
11  private boolean active;
12  private InternalState currentState;
13  private LinkedList<Event> eventPool;
14  private Map<String, Object> memory;
15
16  // Constructors and additional helper functions have been
17  // removed from the sake of clarity
18
19  public void run() {
20    while(active) {
21      Event event = getNextEvent();
22      switch(currentState) {
23      case STATE1:
24        if (event.getSenderId() == 0 && event.getId() == 3) {
25          this.memory.put("v1", event.getParameter("p1"));
26          this.currentState = InternalState.STATE2;
27
28        } else {  /* Illegal Transitions */ }
29        break;
30
31      case STATE2:
32        if (event.getSenderId() == 0 && event.getId() == 3) {
33          Message message = new Message();
34          message.setMessageId(1);
35          message.setParameter("method", memory.get("md"));
36          message.setParameter("parameters", event.getParameter("tuple"));
37          partner_2.sendMessage(message);
38          this.currentState = InternalState.STATE1;
39
40        } else { /* Illegal transitions */ }
41
42        break;
43      default:
44        // Illegal states
45        break;
46      }
47    }
48  }
49
50  private Event getNextEvent() {
51    Event result = null;
52    synchronized (eventPool) {
53      while (eventPool.isEmpty()) {
54        try {
55          eventPool.wait();
56
57        } catch (InterruptedException e) {}
58      }
59      result = eventPool.remove(0);
60    }
61    return result;
62  }
63
64  public void start() {
65    this.active = true;
66    this.internal = new Thread(this);
67    this.internal.start();
68  }
69
70  public void stop() {
71    this.active = false;
72  }
73
74  public void process(Event event) {
75    synchronized (eventPool) {
76      this.eventPool.add(event);  \
77      eventPool.notifyAll();
78    }
```

```
79     }
80 }
```

The *run* operation, which starts at line 19, encapsulates the behavior of the mediator and the two nested switch statements. The task of the mediator thread is to extract messages from the pool and then to process them until there is no more messages to process or until the mediator is stopped. To do so, the mediator first extracts the oldest message from the pool (see line 21) and then process it according to its current state. It is worth to note that the *getNextMessage* operation is actually blocking if there is no message in the pool, avoiding an "active wait" consequently (see line 55).

The first "switch statement" on line 23 selects the proper treatment to apply with respect to the current state of the mediator. Then, the mediator selects the transition to trigger depending on the type of the message that has been extracted and on the partner who emitted it. This second "switch", which is actually an *if-then-else*, appears on lines 24 and 32. While triggering a transition, the mediator may store some data carried on by incoming messages into its own memory (see line 25), or may create new outgoing messages and send them to its partners (as shown on line 33). Finally, the mediator updates its current state by assigning a new value to the *currentState* attribute, as shown on line 26 and 38.

The use of two nested switch statements ensures good performance (especially execution time) but makes impossible the dynamic extension (adding states or events) without regenerating the whole code.

5.3.3 Compiling and Packaging CONNECTors

Figure 5.8 below illustrates the packaging process. In order to be able to automate the complete deployment process (including code compilation, packaging, deployment and starting up), the code generator outputs the following resources:

Java Source File These files are the actual implementation of the CONNECTor model in Java. The current code generation prototype implements CONNECTors using the nested switch strategy illustrated in Section 5.3.2. The code leverages some facilities provided by the OSGi platform, such as logging or service discovery.

Manifest File This file, required to build an OSGi bundle, explicits the dependencies of the CONNECTor on others bundles, services, or resources.

Ant Build File This file describes how to automate the rest of process, especially how to compile, package and deploy the source code that has been generated.

Apart of the code generation itself, the rest of the process is automated using the Ant tool. As mentioned above, the Ant build file contains all the instructions needed for the generation of the final artifact. Figure 5.8 shows the construction of an OSGi bundle, as implemented in the current version of the prototype. It requires first to compile the Java source files, and then to wrap up the manifest file and the resulting class files into a JAR file. This JAR file is the actual artifact that will be later deployed on the OSGi platform.

5.3.4 Deploying and Starting up CONNECTors

The Ant build file that is generated by the code generator also permits to automate the deployment of the resulting OSGi bundle. Recent OSGi platforms such as Apache Felix for instance provide high-level facilities to automate the deployment and the starting up of bundles. Apache Felix comes with a simple *File Install Agent* watching predefined directories. This agent detects new bundles that are written into these directories, and automatically launch and start them.

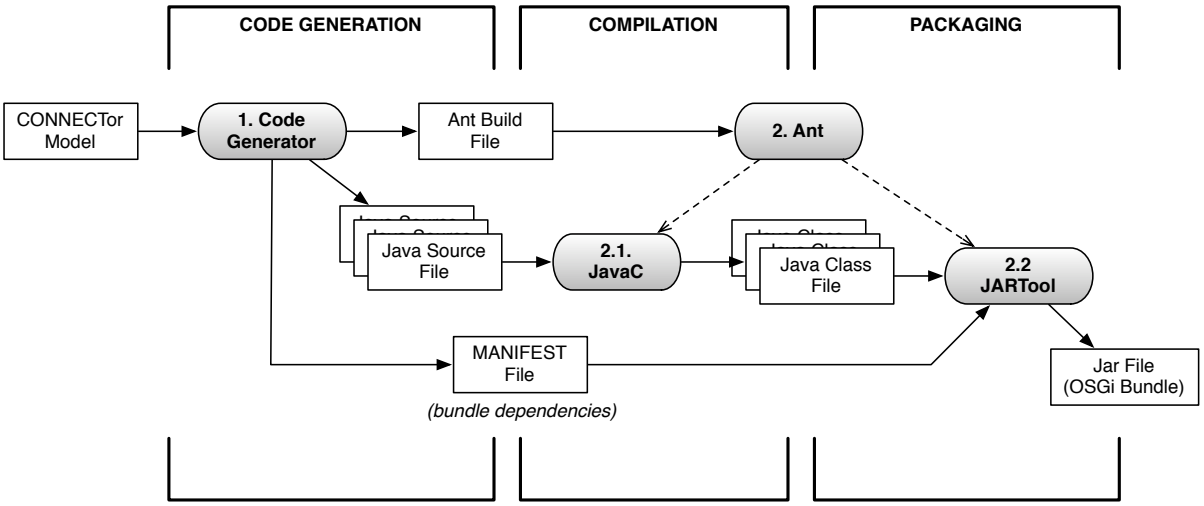


Figure 5.8: Compiling and Packaging CONNECTor source files into an OSGi bundle

It is worth to note that the OSGi platform first resolves bundles' dependencies before to actually start them. Therefore, when a CONNECTor is deployed and started, its dependencies towards the interoperability proxies are automatically and properly resolved by the OSGi platform. This mechanism, combined with the *File Install Agent* previously mentioned, greatly simplifies the automation of the CONNECTor deployment.

5.3.5 Prototype Tool Support

We provide a first prototype implementation of our code generation process that generates CONNECTors as OSGi bundles and deploys them on the Apache Felix platform.

The code generator has been developed using the JET[1] library. Figure 5.9 presents the basic use of the JET libraries. The developer of the code generator first defines a set of templates, which are excerpts of desired output code, where some parts have been replaced by some Java instructions computing the needed contents. The JET engine, then processes these templates and generates a proper Java class for each template. These classes can be later integrated within any traditional Java application.

Each artifact outputted by the code generator (c.f., Figure 5.8) is actually derived from a specific template. The Java source code uses two separate templates: one to generate the actual implementation of the mediator, and another one to generate the ActivatorBundle class, needed to create OSGi bundle. The code generator that outputs the manifest file, as well as the one that creates the Ant build file are also built from two separate templates.

5.4 Towards Model Interpretation

This section introduces the work planned for the year 3 of the project, especially the dynamic interpretation of the CONNECTor model that has been discussed in Section 5.1.3. The first part recalls the basics of service-oriented computing and highlights the similarities and the differences between the CONNECTor model and existing service orchestrations models. The second part illustrates how the code generation techniques presented in Section 5.3.2 can be reused to translate a CONNECTor model into a BPEL process. Finally, the last part outlines how existing BPEL execution engines can be extended in order to support the middleware heterogeneity issue faced by the CONNECT project.

[1] JET stands for Java Emitter Template

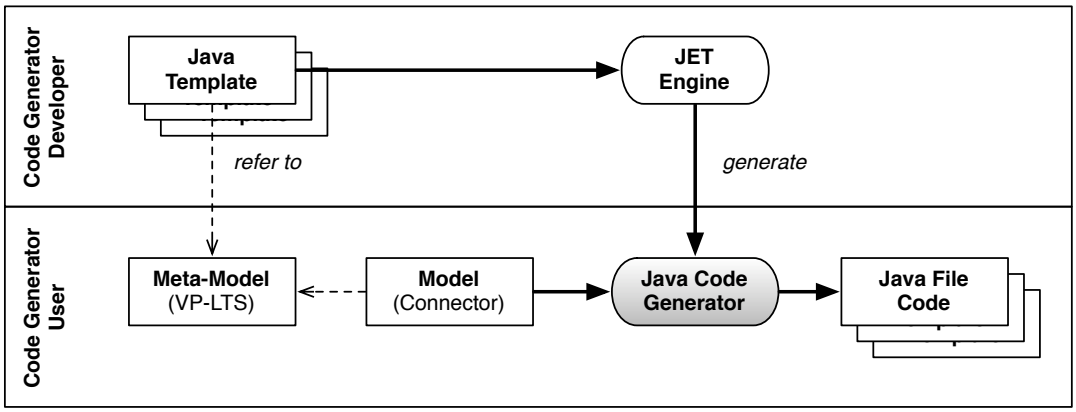


Figure 5.9: Building the code generator from code templates using JET

5.4.1 CONNECTors vs. Service Orchestrations

Besides the generation of imperative code, it is also possible to dynamically interpret the model. The idea behind interpretation is to embed the CONNECTor model within the run-time CONNECTor, and to dynamically consult it, in order to know how to handle incoming messages. As explained previously, this alternative method provides more flexibility: any change in the CONNECTor model will be immediately effective. The CONNECTor model properly encapsulates the expected behavior, whereas the CONNECTor engine, in charge of interpreting the CONNECTor model, can consequently be reused from one CONNECTor to another.

The *Service-Oriented Architecture* paradigm [41, 18, 61] advocates to structure software systems as a composition of loosely coupled services. Each service, which encapsulates an independent functionality, can be combined with other services to build a higher level service, which can be composed the same way. As explained by Peltz [70], compositions of services are currently envisioned in two ways: service orchestrations and service choreographies.

Service Orchestrations are centralized compositions of services. The composition is modeled as a flow of data between a set of external services. The idea behind a service orchestration is to execute the resulting workflow on a single point, calling external services, and performing all the needed data conversions.

Service Choreographies are distributed compositions of services. Each service involved in the choreography executes a part with respect to the behavior of other participants, without any central coordination.

Although service orchestrations are now commonly used, whereas choreographies remain more academic issues, these two approaches are not conflicting but are rather two different views of the composition of services. Intuitively, there are a lot of similarities between CONNECTor as modeled in Section 5.2 and service orchestrations. Both CONNECTors and service orchestrations:

- coordinate behavior of several networked systems ;
- formalize the possible sequences of messages for a given purpose ;
- must support data mismatches in order to enable real interoperability between networked systems.

The main difference between these two notions arises at the technical level: While implementing service orchestrations with existing technologies, one assumes that all the networked systems at play in the collaboration are using the same middleware technology, namely Web

Services. Therefore, the notion of CONNECTor discussed here is indeed a generalization of service orchestrations.

The main technology enabling service orchestrations is the BPEL language [64], which captures service processes in a block-structured language. BPEL processes can be interpreted by any engine compliant with the BPEL standard [64], such as Active BPEL, Oracle Business Process Manager, or Apache ODE for instance. It is worth to note that BPEL is now the *de facto* standard used by both academics and industries to describe business processes.

5.4.2 From CONNECTors to BPEL Processes

Although BPEL processes and CONNECTor models may seem very similar, especially when graphically depicted as finite state machines, there are some fundamental differences that prevent the use of a direct mapping between these two formalisms. This section illustrates how the code generation strategy presented in Section 5.3.2 can be used to generate a relevant BPEL process providing the same behavior than the CONNECTor model.

Intuitively, a CONNECTor model and a business process are somehow similar: both can be seen as graphs labeled with actions. If this holds for abstract process languages such as BPMN, it does not for BPEL, which is not a graph-based language but a block-structured language, whose programs are sequences of statements including service invocations, loops, or conditional branches.

The translation from graph-based to block-structured formalisms therefore implies to translate the control flow of the source graph into a sequence of statements so that the traces of the resulting program match all the possible paths of the source graph. This is a difficult problem, especially if the source graph contains some *ill-formed nested loops*: loops that starts within another loop but ends outside of this other loop (or vice versa). The removal of these ill-formed loops is similar to the *GOTO removal problem*, where given a program containing many *go to* statements, one tries to generate a program having the same behavior but described only using higher level loops such as *while* or *for*. Solutions yet exists such as [57, 67] but remain complex and time consuming and consequently irrelevant for the dynamic generation.

A simpler solution is to make explicit the control flow using a variable representing the current state and conditional branching to implement the CONNECTor model transitions. This solution is exactly similar to the application of the solution 1 of Table 5.1. Assuming that the current state of the CONNECTor model can be stored as a variable (say as an integer value), a first ”switch” statement can be used to branch between possible values of the current state. Within each of these possible branches, another switch statement is used to branch over possible incoming events (i.e., incoming messages).

The following BPEL code illustrates the implementation of such a solution. With respect to the BPEL terminology, the conditional branching with respect of the current state appears as a `if` activity. By contrast, the conditional branching on incoming message are realized using a `pick` activity. In each case, the local variable representing the current state of the LTS is assigned with a new value, reflecting the transition between two states of the LTS.

```
<process name="mediator4LimeAndSoap">
  <extensions>
    <extension namespace="http://www.connect.eu/lime"
      mustUnderstand="yes" />
    <extension namespace="http://www.connect.eu/soap"
      mustUnderstand="yes" />
  </extensions>

  <variables>
    <!-- Current State of the LTS -->
    <variable name="currentState" type="xsd:integer" />
    <variable name="stopped" type="xsd:boolean" />

    <!-- Variables needed to propagate data -->
    <variable name="info" type="stadium:info" />
    <variable name="request" type="stadium:request" />
```

```
17  </variables>
18
19  <while>
20    <condition>$stopped != false</condition>
21    <if>
22        <condition>
23          bpel:getVariable('currentState') == 1
24        </condition>
25      <connect:pick>
26        <lime:onMessage partnerLink="customer"
27                    operation="rdg"
28                    variable="getInfo">
29          <!-- Invoke the search on the merchant side-->
30        <soap:invoke name="purchase" partnerLink="merchant"
31                operation="MSEARCH"
32              inputVariable="info"
33          </invoke>
34        <!-- Update the current state -->
35        <assign>
36          <copy>
37            <from>
38              <literal>2</literal>
39                </from>
40                <to variable="currentState" />
41          </copy>
42        </assign>
43         </lime:onMessage>
44       <lime:onMessage partnerLink="customer"
45                operation="out"
46            variable="request">
47        <!-- No special action required for this event -->
48        <assign>
49          <copy>
50            <from>
51              <literal>3</literal>
52                  </from>
53                  <to variable="currentState" />
54          </copy>
55        </assign>
56            </lime:onMessage>
57      </connect:pick>
58        <elseif>
59              <condition>
60          bpel:getVariable('currentState') == 2
61        </condition>
62        <connect:pick>
63          <soap:onMessage partnerLink="merchant"
64                      operation="response"
65                      variable="info">
66            <lime:invoke name="in" partnerLink="customer"
67                operation="in"
68              inputVariable="info"
69            </lime:invoke>
70          <assign>
71            <copy>
72              <from>
73                <literal>1</literal>
74                    </from>
75                    <to variable="currentState" />
76            </copy>
77          </assign>
78          </soap:onMessage>
79        </connect:pick>
80        </elseif>
81        <elseif>
82              <condition>
83          bpel:getVariable('currentState') == 3
84        </condition>
85        <connect:pick>
```

```
86          <lime:onMessage partnerLink="customer"
87                operation="reactTo"
88                      variable="req">
89                   <!-- Invoke the search on the merchant side-->
90            <soap:invoke name="purchase" partnerLink="merchant"
91                    operation="requestFor"
92                  inputVariable="req"
93            </invoke>
94            <!-- Update the current state -->
95            <assign>
96              <copy>
97                <from>
98                  <literal>1</literal>
99                      </from>
100                     <to variable="currentState" />
101           </copy>
102           </assign>
103            </lime:onMessage>
104       </connect:pick>
105       </elseif>
106       <else>
107       <throw faultName="lts:IllegalState" />
108       </else>
109    </if>
110  <while>
111 </process>
```

It is worth to note the use of the extensions previously mentioned. The pick activities are specific to the connect project, as show their name which is prefixed by ”connect”. Within these activities, invocations of remote services depend on specific middleware technologies, which appear as namespaces as well (e.g., lime:invoke). BPEL 2.0 provides an extension mechanism enabling the definition of new activities which can support the example described above. This approach would preserve the simplicity of the mediation process as well as the separation of concerns between the specification of the mediation and its realization. However, this would make the BPEL code not compliant with standard BPEL engines and consequently calls for the modification of existing ones. A better solution would be to describe the mediation using standard BPEL activities, regardless of the middleware technologies used by the partners, and to push all the ”intelligence” into the interpretation engine. This would make BPEL a real service orchestrations language, and not a web-service orchestrations language.

5.4.3 Towards a Multi-Protocols BPEL Engine

In order to support the interpretation of BPEL processes involving partners using versatile middleware technologies, we envision to extend the Apache ODE BPEL engine. Figure 5.10 shows the internal architecture of this engine. The ODE engine first compiles a given BPEL process into a internal representation that better fits the need of runtime execution. This compiled version of the process is actually executed by the *ODE BPEL Runtime*. The execution of a BPEL process requires two main features: the interaction with external web-services and the access to external data-bases. The first one is handled by a module entitled *Jacob*, which actually delegates the invocation of web-services to an integration layer (based on the AXIS library).

As shown on Figure 5.10, the assumption that all partners are actually web-services shows through the architecture of the Apache ODE engine: the integration layer only supports this specific technology. In order to enable the integration of partners that are not deployed as web services, it is necessary to replace the original integration layer by a new one able to interact with various technologies. This solution is illustrated on Figure 5.11 where a new integration layer supports (via a system of plug-ins) various middleware technologies. The idea beyond this system of plug-ins, is to reuse the interoperability proxies that have been used for the code generation (see Section 5.1.1) and to connect them with a specific integration layer, able to switch among them.

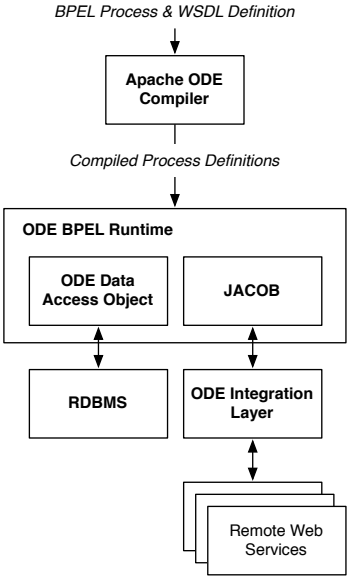


Figure 5.10: Internal Architecture of the BPEL Apache ODE Engine

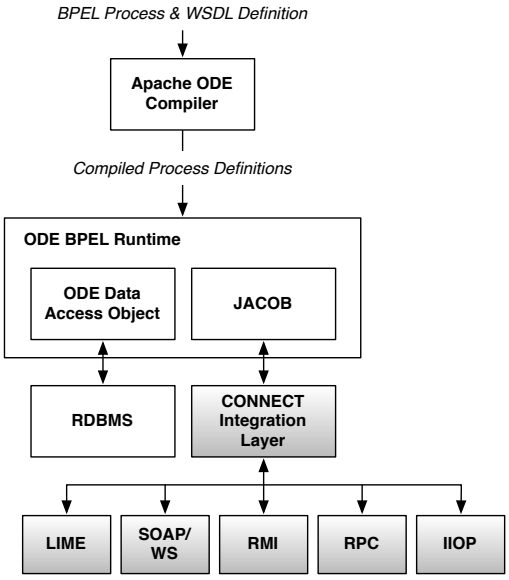


Figure 5.11: A possible extension of the Apache ODE Engine to support communication with various middeware technologies

5.5 Application to the Photo Sharing Scenario

This last section illustrates the generation of Java code using the photo sharing example introduced in the previous chapters. For the record, we consider two applications providing similar functionality (sharing photos) but which are different at both the middleware and the application levels. At the middleware level, the first application is implemented over the LIME middleware, which provides a distributed shared memory. By contrast, the second application is implemented using SOAP-RPC technology, where communications are based on synchronous operation invocations. In addition, at the application level, the first application expects a photo to be transmitted in two parts, the meta-data describing its contents, and then, in a separate message, the data of the related file. The SOAP-RPC application expects a photo to be transmitted in one single message containing both the data and the meta-data. These differences are illustrated on Figures 5.12 and 5.13 below, which model the behavior of the LIME-based photo producer and the behavior of the SOAP-RPC photo server, respectively.

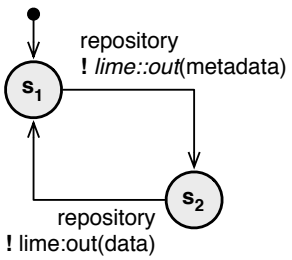


Figure 5.12: Expected behavior of the LIME photo producer

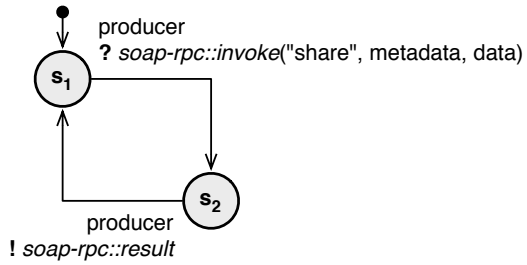


Figure 5.13: Expected behavior of the SOAP-RPC server

In these two figures, we denote the emission of a message $m(p_1, \ldots, p_n)$ to a given partner p by $p\,!\,m(p_1, \ldots, p_n)$ and the reception of a message from this partner by $p\,?\,m(p_1, \ldots, p_n)$.

In Figure 5.12, the producer sends two messages containing two different parts of the same picture, and does not wait for any kind of acknowledgment, whereas on Figure 5.13 the SOAP server expects only one message, but emits a confirmation result. These two LTS (Figures 5.12 and 5.13) are the main inputs of the synthesis process, which outputs the needed mediation behavior depicted on Figure 5.14.

The needed mediation implies to first receive the meta-data sent by the producer and to store them into a local memory (the assignment is denoted by $v_1 \leftarrow metadata$). Then, we receive the second part of the picture, namely the photo data and the mediator consequently sends the proper SOAP-RPC request containing both the data and the meta-data to server. Finally, the mediator intercepts the result emitted by the server but does not forward it to the producer, since he does not expect any.

Before to generate code from the LTS model shown by Figure 5.14, we refine it by building the equivalent mealy machine, as shown on Figure 5.15. For each message that is received, this transformation fold the sequence of resulting actions into the same transition, consequently reducing the number of states.

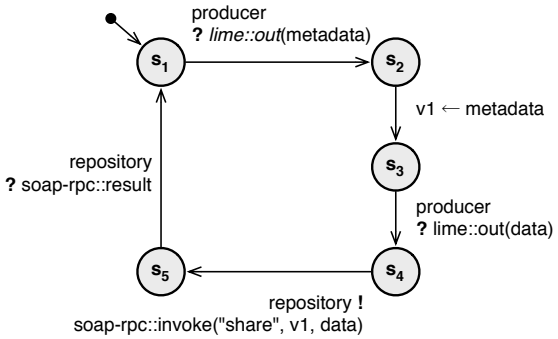


Figure 5.14: LTS resulting from the synthesis and capturing the needed behavior to properly mediate between the LIME producer and the SOAP server

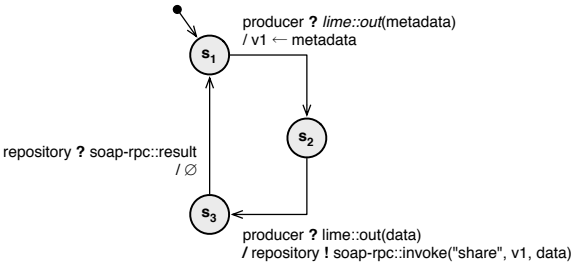


Figure 5.15: Equivalent Mealy machine, representing the mediation behavior

The following code excerpt shows the part of the generated code that implements the mediation logic. The implementation of the first transition of the mealy machine appears on line 17: it shows how data can be extracted from incoming messages and then stored in local variables. It is worth to note that the interoperability proxies are in charge of parsing LIME messages and to build abstract messages that can be manipulated independently of the protocol. Line 25 shows the second transition, and illustrates how data can be read from the memory and then combined to build a relevant message. Here too, the proxy is responsible for the proper construction of the real SOAP-RPC invocation that will be emitted on the network. The last transition, which is implemented on line 38, merely changes the current state of the automaton.

```
1  public class PhotoSharingMediator implements Mediator {
2
3    private enum InternalState {
4      STATE1, STATE2, STATE3,
5    }
6
7    private Proxy producer;
8    private Proxy repository;
9
10   // [...]
11
12   public void run() {
13     while(active) {
14       Event event = getNextEvent();
```

```
15        switch(currentState) {
16        case STATE1:
17          if (event.getSenderId() == 1 && event.getId() == 1) {
18            this.memory.put("v1", event.getParameter("metadata"));
19            this.currentState = InternalState.STATE2;
20
21          } else {  /* [...] */ }
22          break;
23
24        case STATE2:
25          if (event.getSenderId() == 1 && event.getId() == 2) {
26            Message message = new Message();
27            message.setMessageId(1);
28            message.setParameter("operation", "share");
29            message.setParameter("parameter", memory.get("v1"));
30            message.setParameter("parameter", event.getParameter("data"));
31            repository.sendMessage(message);
32            this.currentState = InternalState.STATE3;
33
34          } else { /* Illegal transitions */ }
35
36          break;
37        case STATE3:
38          if (event.getSenderId() == 2 && event.getId() == 2) {
39            this.currentState = InternalState.STATE1;
40
41          } else { /* Illegal transitions */ }
42
43          break;
44        default:
45          // Illegal states
46          break;
47        }
48      }
49    }
50
51    // [...]
52  }
```

As explained on Figure 5.8 page 76, two additional artifacts are generated to help compiling the code: an Ant build file and a manifest file. The following code snippet shows the complete manifest file that is generated for the photo sharing example. The dependencies of the resulting OSGi bundle are made explicit on line 10.

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: eu.connect.osgi.photosharingSimple
4  Bundle-SymbolicName: eu.connect.osgi.photosharingSimple
5  Bundle-Version: 1.0.0.qualifier
6  Bundle-Activator: Activator
7  Import-Package: org.osgi.framework;version="1.3.0",
8   org.osgi.service.log;version="1.3.0"
9  Bundle-RequiredExecutionEnvironment: J2SE-1.5
10 Require-Bundle: eu.connect.osgi.proxyfactory;bundle-version="1.0.0",
11  eu.connect.osgi.connector;bundle-version="1.0.0"
```

The Ant build file, which is generated, is shown below. It defines three rules to automate the packaging process. The rule to compile the generated Java code is shown on line 21 ; the rule to package the resulting Java class files into a Jar file is shown on line 35 ; and the rule to deploy the final OSGi bundle appears on line 41 (Further details about the deployment mechanisms provided by the OSGi platform can be found in the deliverable D1.2). It is worth to note that

the values of most of the properties that appear in this Ant build file (see line 7) depend on the machine where the code generator is run, but can be set up in a separate configuration file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="PhotoSharing Simple" default="compile">
3    <description>
4      Build and deploy the connector PhotoSharing Simple
5    </description>
6
7    <property name="src.package" value="eu.connect.osgi" />
8    <property name="src.dir" location="src" />
9    <property name="bin.dir" location="bin" />
10   <property name="bundle.dir" location="/Users/franck/Documents/dev/plugins" />
11   <property name="osgi.dir" location="/Applications/Eclipse 2/plugins/" />
12   <property name="osgi.jar" value="org.eclipse.osgi_3.6.1.R36x_v20100806.jar" />
13   <property name="bundle.name" value="${src.package}.${ant.project.name}_1.0.0.jar" />
14
15   <!-- Create the ${bin} directory if it does not exits -->
16   <target name="init">
17     <mkdir dir="${bin}" />
18   </target>
19
20   <!-- Compile the source files found in ${src} into ${bin} -->
21   <target name="compile" depends="init" description="Compilation of the connector source
         files">
22     <javac srcdir="${src.dir}" destdir="${bin.dir}" debug="off">
23       <classpath>
24         <fileset dir="${osgi.dir}">
25           <include name="${osgi.jar}" />
26         </fileset>
27         <fileset dir="${bundle.dir}">
28           <include name="**/*.jar" />
29         </fileset>
30       </classpath>
31     </javac>
32   </target>
33
34   <!-- Build the OSGi bundle ${bundle.name} -->
35   <target name="build" depends="compile" description="Packaging of the binary files as an
         OSGi bundle">
36     <jar basedir="${bin.dir}" destfile="${bundle.name}" manifest="META-INF/MANIFEST.MF">
37     </jar>
38   </target>
39
40   <!-- Deploy the OSGi bundle -->
41   <target name="deploy" depends="build" description="Deploying the OSGi bundle">
42     <copy todir="${bundle.dir}">
43       <fileset dir=".">
44         <include name="${bundle.name}" />
45       </fileset>
46     </copy>
47   </target>
48
49 </project>
```

5.6 Conclusion

This chapter presented the last step of the synthesis process, namely the construction of an executable artifact that can be automatically deployed and run. This final artifact results from the combination of a mediator component with a set of proxies, enabling communications with different middleware technologies. The mediator part, which is application-dependent, is dynamically obtained from the behavioral models outputted by the previous steps of the synthesis process,

whereas the proxies are reused from one CONNECTor to another and can be configured from a description of the middleware technologies.

We envisioned two main approaches to realize a CONNECTor: the generation of code and the dynamic interpretation of the CONNECTor model. These two approaches vary in both performance and flexibility: the code generation ensures better performance but less flexibility than the interpretation.

Regarding code generation, we first surveyed existing code generation strategies and we developed an initial prototype to support the generation of Java code, its compilation and the packaging into an OSGi bundle that can be directly deployed. We illustrated its use using the Photo Sharing example that has been previously introduced.

The dynamic model interpretation remains in its infancy and will be further investigated during the third year of the project. As we leveraged the similarities between CONNECTor models and service orchestrations, we plan to extend existing service orchestrations engines to support the composition of heterogenous services (and not only web-services).

6 Dealing with Non-Functional Properties

To build an interoperability solution between the networked systems populating the environment, two aspects have to be considered: *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). Indeed, while building an interoperability solution, both functional and non-functional properties of the connected system under-construction must be taken into account and ensured.

Towards this direction, this Chapter shows a methodology to secure the synthesized CONNECTors. In particular, the approach we propose is independent from the specific synthesis technique used for generating the CONNECTor. It is based on *modal logic* and *partial model checking* in order to automatically generate the minimum controller for guaranteeing that the CONNECTor execution is secure.

Let us start by recalling some results about partial model checking.

6.1 Compositional analysis: the *partial model checking*

When a complex system has to be developed, the developer considers it as a combined system and tries to understand which properties each subsystem has to satisfy in order to guarantee that the combined system works as expected. Indeed, usually, the implementation cannot be immediately built based on specification from the specification, the implementation phase consists of a large number of small refinements of the initial specification until eventually the implementation can be clearly identified. [9] has proposed the *partial model checking* mechanism in order to give a compositional method for proving properties of concurrent systems, *i.e.,* the task of verifying an assertion for a composite process is decomposed into verification tasks for the sub-processes.

The intuitive idea underlying the partial model checking is the following: proving that $P \| Q$, where $\|$ is the process algebra parallel operator, satisfies an equational μ-calculus formula ϕ is equivalent to prove that Q satisfies a modified specification $\phi_{//_P}$, where $//_P$ is the *partial evaluation function* for the parallel composition operator (Figure 6.1).

First of all, we briefly recall some notions about equational μ-calculus. It is based on *fix-point equations* that allow to define recursive properties of a system.

Let Act be the set of actions ranged over by $a, b, \ldots$, let Z be a variable ranging over a set of variables V. The syntax of the assertions (ϕ) and of the lists of equations (D) is given by the following grammar:

$$\phi ::= Z \mid \mathbf{T} \mid \mathbf{F} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a \rangle \phi \mid [a]\phi$$

$$D ::= Z =_\nu \phi, D \mid Z =_\mu \phi, D \mid \epsilon$$

where $\mathbf{T}$ and $\mathbf{F}$ are the logical constants, respectively, *true* and *false*; $\wedge$ and $\vee$ are the classical symbols for conjunction and disjunction. The possibility modality $\langle a \rangle \phi$ expresses the ability to have an a transition into a state that satisfies ϕ. The necessity modality $[a]\phi$ expresses that after each a transition there is a state that satisfies ϕ. The minimal (maximal) fix-point equation syntax, $Z =_\mu \phi$ ($Z =_\nu \phi$), is used to define recursive formulas. It is assumed that variables appear only once on the left-hand sides of the equations of the list, the set of these variables being denoted as $Def(D)$. A list of equations is closed if every variable that appears in the assertions of the list is in $Def(D)$. The semantics of equational μ-calculus is given by means of LTS.

The meaningful result of the partial model checking is the following lemma.

Lemma 1 *[9] Given a process $P \| Q$ (where P is a finite-state process) and an equational specification $D \downarrow Z$ we have:*

$$P \| Q \models (D \downarrow Z) \text{ iff } Q \models (D \downarrow Z)_{//_P}$$

$$(D\downarrow Z)/\!/t = (D/\!/t)\downarrow Z_t$$

$$\epsilon/\!/t = \epsilon$$

$$(Z =_\sigma \phi D)/\!/t = ((Z_s =_\sigma \phi//s)_{s\in S})(D)/\!/t$$

$$Z/\!/t = Z_t$$

$$\phi_1 \wedge \phi_2/\!/s = (\phi_1/\!/s)\wedge(\phi_2/\!/s)$$

$$\phi_1 \vee \phi_2/\!/s = (\phi_1/\!/s)\vee(\phi_2/\!/s)$$

$$[a]\phi/\!/s = [a](\phi/\!/s)\wedge \bigwedge_{s\xrightarrow{a}s'} \phi/\!/s', \text{ if } a \neq \tau$$

$$\langle a\rangle\phi/\!/s = \langle a\rangle(\phi/\!/s)\vee \bigvee_{s\xrightarrow{a}s'} \phi/\!/s', \text{ if } a \neq \tau$$

$$[\tau]\phi/\!/s = [\tau](\phi/\!/s) \wedge \bigwedge_{s\xrightarrow{\tau}s'} \phi/\!/s' \wedge \bigwedge_{s\xrightarrow{a}s'} [\overline{a}](\phi/\!/s')$$

$$\langle\tau\rangle\phi/\!/s = \langle\tau\rangle(\phi/\!/s) \vee \bigvee_{s\xrightarrow{\tau}s'} \phi/\!/s' \vee \bigvee_{s\xrightarrow{a}s'} \langle\overline{a}\rangle(\phi//_{s'})$$

$$\mathbf{T}/\!/s = \mathbf{T}$$

$$\mathbf{F}/\!/s = \mathbf{F}$$

Figure 6.1: Partial evaluation function for parallel operator, where $t = (S, Act, \rightarrow)$ is the labelled transition system representing a process while $s \in S$ is the state in which we evaluate the formula [9].

This lemma allows to partially evaluate an equational μ-calculus formula with respect to the behaviour of a component of the considered system (P in the lemma), in such a way that the reduced formula $\phi_{//P}$ depends only on the formula ϕ and on process P. No information is required about the process Q which can represent a possible malicious component.

Remarkably, this function is exploited in [9] to perform model checking efficiently, where both P and Q are specified. In our setting, the process Q is not specified. Thus, given a certain system P, it is possible to find the property that the malicious component must satisfy to avoid a successful attack on the system. It is worth noticing that partial model-checking function may be automatically derived from the semantics rules used to define a language semantics. Thus, the proposed technique is very flexible. According to [9], when ϕ is *simple*, *i.e.,* it is of the form X, $\mathbf{T}$, $\mathbf{F}$, $X_1 \wedge \ldots \wedge X_k \wedge [a_1]Y_1 \wedge \ldots \wedge [a_l]Y_l$, $X_1 \vee \ldots \vee X_k \vee \langle a_1\rangle Y_1 \vee \ldots \vee \langle a_l\rangle Y_l$, then the size of $\phi_{//P}$ is bound by $|\phi| \times |P|$. Any assertion can be transformed to an equivalent simple assertion in linear time. Hence, we can conclude that the size of $\phi_{//P}$ is polynomial in the size of ϕ and P.

It is important to notice that a lemma similar to Lemma 1 holds for each process algebra operator. This is a meaningful result because it allows us to evaluate a formula according to a part of a system, whatever structure the system may have, *i.e.,* independently from how the different components of the system are related to one another.

6.2 Securing the CONNECTor by Differential Controller

Here, we propose an approach based on process algebra *controller operators* [54] for controlling the behaviour of a synthesized functional CONNECTor in order to guarantee that it has been executed in a secure way, *i.e.,* according to the Networked Systems (NSs for short) security requests.

The approach we propose here is mainly based on

- we use *modal logic* and *satisfiability procedure* for synthesizing process, in this case *controller programs*. LTS can be used for expressing the semantics of the controller program but the security property we want to satisfy is expressed as a modal logic.
- we use the *partial model checking* technique for pointing out the necessary and sufficient conditions that a controller programs has to satisfies in order to guarantee a certain security properties under investigation.

We assume that each NS asks to the CONNECT infrastructure for establishing a communication, send to the CONNECT infrastructure, in particular to the *synthesis enabler*, not only their functional requirements and their description (ref. to Discovery enabler) but also their security requirements. Let us also assume that these security requirements are expressed as security properties, hereafter denoted by P_1 and P_2, that state the security policies of each NS. Let us suppose that such security requests are expressed as an *equational* μ-calculus formula [34].

Our framework is based on *process algebra*, *e.g.*, [39, 56], *partial model checking* [9] and *open system paradigm* suggested for the modelling and the verification of systems [52, 53], and here extended for dealing with the synthesis of controller program able to assure that the running CONNECTor satisfies both P_1 and P_2.

Using this formal approach for securing synthesized CONNECTors provides several advantages. One of the main advantages is the modularity of the approach. Indeed, we are able to synthesize appropriate controller program according to the semantics definition of differential controller on top of existing mediator. This implies that:

- It is not necessary to re-synthesize the whole CONNECTor if only security requirements change.
- We are able to adapt existing CONNECTor to new request.

Moreover, the proposed approach, thorough the application of the partial model checking technique, allows to automatically generate the minimum controller program that, composed with the synthesized CONNECTor, satisfies the both functional and security requirements for the communication.

Let us consider the CONNECTor as the system component that has to be made secure. Hence, the problem we address is the following one:

$$NS_1 \| C \| NS_2 \models \phi \tag{6.1}$$

where ϕ is a logic formula expressing the security property and C the synthesize CONNECTor. If it does not hold, we introduce a controller operator $\triangleright$ and describe a mechanism for synthesizing a *differential controller* program Y that, according to the semantics of the controller operator, assures that the considered system results secure. There is not a unique way to control a target system in order to enforce security properties. It is possible to use a different controller operator, according to which properties the system has to satisfy and how Indeed, it is possible to define several controller operators with different behaviours.

According to the edit automata definition given in [73, 12], we define four different controller operators [55]: $Y \triangleright_T X$, $Y \triangleright_S X$, $Y \triangleright_I X$ and $Y \triangleright_E X$ where X generically denotes the application that is controlled by Y and T stands for *Truncation*, S for *Suppression*, I for *Insertion* and E for *Edit*.

The semantics definitions of those four operators are recalled in Table 6.1.

The semantics rule of the truncation operator $\triangleright_T$ states that if E and F perform the same action, thus such action is allowed. Hence, the controlled process $E \triangleright_T F$ performs the action a, otherwise it halts. It is easy to note that this operator is similar to the parallel composition defined in Chapter 4.

Same as for the truncation automaton, the semantics rules of the suppression operator $\triangleright_S$ state that if F performs the same action performed by E also $E \triangleright_S F$ performs it. On the contrary, if F performs an action a that E does not perform and E performs the control action $-a$ then

Truncation:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd_T F \xrightarrow{a} E' \rhd_T F'}$$

Suppression:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd_S F \xrightarrow{a} E' \rhd_S F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \rhd_S F \xrightarrow{\tau} E' \rhd_S F'}$$

Insertion:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \rhd_I F \xrightarrow{a} E' \rhd_I F'} \qquad \frac{E \xrightarrow{a} \not\; E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \rhd_I F \xrightarrow{b} E' \rhd_I F}$$

Edit:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{a} E' \rhd_E F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{\tau} E' \rhd_E F'}$$

$$\frac{E \xrightarrow{a} \not\; E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{b} E' \rhd_E F}$$

Table 6.1: Semantics definition of controller operators for enforcing safety properties.

$E \rhd_S F$ performs the action τ that *suppresses* the action a, *i.e.,* a becomes not visible from external observation. Otherwise, $E \rhd_S F$ halts.

Similarly, the semantics rules of the insertion operator $\rhd_I$ state that if F performs an action a that also E can perform, the whole system makes this action. If F performs an action a that E does not perform and E detects it by performing a control action $+a$ followed by an action b, then the whole system performs b. It is possible to note that in the description of insertion automata in [12] the domains of γ and δ are disjoint. In our case, this is guaranteed by the premise of the second rule, *i.e.,* $E \xrightarrow{a} \not\; E'$, $E \xrightarrow{+a.b} E'$. In fact, if a pair (a, q) is not in the domain of δ and it is in the domain of γ, it means that, in the state q, a actions cannot be performed. Thus, in order to change state, an action different from a must be performed. It is important to note that the controller is able to insert new actions, but it is not able to suppress any action performed by F.

In order to have the capabilities of both insertion and suppression together, we define the controller operator $\rhd_E$. Its rules are the union of the rules of the $\rhd_S$ and $\rhd_I$.

We use controller operators in such a way that the specification of the system becomes:

$$\exists Y \quad NS_1 \| (Y \rhd_{\mathbf{K}} C) \| NS_2 \models \phi \tag{6.2}$$

By using *partial model checking* [9] it is possible to reduce it as follows:

$$\exists Y \quad Y \rhd C \models \phi' \tag{6.3}$$

where $\phi' \cong \phi // NS_1 \| NS_2$. Indeed, the *partial model checking* technique allows us for projecting the behaviour of the known part of the system directly into the formula in order to pointed out the necessary and sufficient conditions that the CONNECTor has to satisfy in order to be secure, *i.e.*, to satisfy the security requirements expressed by the logic formula ϕ. Such condition is expressed by ϕ'.

Hereafter we present a possible solution to the problem described in 6.3 and a method for synthesizing a controller program for the given controller operator in order to guarantee *safety properties*, *i.e.*, such properties that state that nothing bad happens.

According to [19], this kind of properties can be expressed by a sub-classes of equational μ-calculus formulas that we call Fr_μ. It consists of equational μ-calculus formulas without $\langle _ \rangle$ operator. It is easy to prove that, according to the rule of the partial evaluation function with respect to parallel operator, this set of formulas is closed under the partial model checking function. Moreover, for this class of formulas, the following result holds.

Proposition 1 ([22]) *Let E and F be two processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$, where $\preceq$ is a* weak simulation *relation.*

In order to define a weak simulation, we recall the following notation. Let us consider $a \neq \tau$, $\hat{a} = a$, and $\hat{\tau} = \epsilon$. Then, notation $P \stackrel{\tau}{\Longrightarrow} P'$ denotes that P and P' belongs to the reflexive and transitive closure of τ. The same holds for notation $P \stackrel{\epsilon}{\Longrightarrow} P'$. Also, $P \stackrel{\hat{a}}{\Rightarrow} P'$ if $P \stackrel{\epsilon}{\Rightarrow} P_\epsilon \stackrel{\hat{a}}{\rightarrow} P'_\epsilon \stackrel{\epsilon}{\Rightarrow} P'$ where P_ϵ and P'_ϵ denote intermediate states[1].

Definition 10 *A relation $\mathcal{R}$ between states of a LTS $L = (\mathcal{E}, Act, \rightarrow)$ is a* weak simulation *if for each $(P, Q) \in \mathcal{R}$ and for each $a \in Act$:*

$$\textit{if } P \stackrel{a}{\longrightarrow} P' \textit{ then there exists } Q' : Q \stackrel{a}{\Longrightarrow} Q' \textit{ and } (P', Q') \in \mathcal{R}.$$

Now, let us consider the following mild assumption on the controller operators we intend to use.

Assumption 1 *For every C and Y, we have:*

$$Y \triangleright C \preceq Y$$

Since we consider formulas in Fr_μ, if a controller operator semantics is defined in such a way that the resulting *controlled program* $Y \triangleright C$ is simulated by the *controller program* Y itself, then we are able to abstract from the universal quantification on C. In fact, in order to solve the problem in Statement (6.3) it is sufficient to solve the following reduced one:

$$\exists Y \quad Y \models \phi' \tag{6.4}$$

The formulation (6.4) is easier to manage than the previous one. In particular, in this way, we have reduced a validity problem to a satisfiability one in μ-calculus. Hence a possible model Y for ϕ' can be found according to the following theorem

Theorem 1 ([77]) *Given a formula ϕ, it is possible to decide within exponential time in the length of ϕ if there exists a model of ϕ and it is also possible to give an example of such model.*

Consequently, we are able to prove the following result

Theorem 2 *Under Assumption 1, the problem described in Formula (6.3) is decidable.*

Note that the trivial solution exists. As a matter of fact the process $\mathbf{0}$ is a model for all possible formulas in Fr_μ, *i.e.*, for every process P, $0 \preceq P$, hence, according to the Proposition 1, $0 \models \phi$. Obviously this is the easiest solution, however, it is possible to find more complex model for ϕ by exploiting satisfiability procedure, *e.g.*, the one developed by Walckiewicz in [83].

Proposition 2 *[55] For the controller operator $\triangleright_{\mathbf{K}}$ defined in Table 6.1 the Assumption 1 holds,* i.e.*,*

$$Y \triangleright_{\mathbf{K}} C \preceq Y$$

Using this approach we are able to enforce safety properties. Example of safety properties are:

Access-Control property The set of proscribed partial executions contains those partial executions ending with an unacceptable operation being attempted. There is no way to "unaccess" the resource and fix the situation afterward. An example of an access-control policy is the *Chinese Wall* policy: Assuming that there are two disjoint sets of resources, once one accesses a resource of one set, it is not possible to access the other set of resources any more and vice-versa.

[1] We can use the short notation $P \stackrel{\epsilon}{\Longrightarrow} \stackrel{\hat{a}}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} P'$ when the intermediate states are not relevant.

Bounded Availability properties may be characterized as safety ones. An example is "one principal cannot be denied the use of a resource for more than D steps as a result of the use of that resource by other principals". Here, the defining set of partial executions contains intervals that exceed D steps and during which a principal is denied use of a resource.

Several other example of safety properties can be provided as, for instance, a property for expressing the possibility to open a new file only if the previous one is closed or that only actions in a given set can be performed by a process in any reachable state.

6.2.1 Application to the Photo-Sharing scenario

Let us suppose to be in the case of and end-to-end application of the Photo-Sharing application. Let us assume that one of the two NS has set the following security requirement.

ϕ="The photo sent must be signed"

Let us suppose to already have the the CONNECTor C that solves or functional mismatching. We have to find Y s.t.

$$Y \triangleright_E C \models \phi$$

The final result is that, even if the NS that is going to send the photo is not able to sign the photo, the controller program Y meets this request by adding to the CONNECTor the ability of signing the photo.

6.3 Conclusion and Future Work

Here we have showed an approach based on process algebra, modal logic, and partial model checking for automatically generating controller programs able to guarantee that a given CONNECTor is secure, *i.e.,* that the CONNECTor satisfies security requirements of Networked Systems.

As future work, we aim to extend our approach for dealing with cryptography. Indeed, we plan to use a process algebra-based formalism able to express also cryptographic primitive in such a way that we are able to synthesize controller program able to add and control also encryption and decryption actions. This leads to be able to enforce at run-time (WP5) also these kind of actions in such a way that we will be able to check and guarantee also cryptographic requirements.

7 Conclusion and Future Works

One of the core challenges of CONNECT is to automatically synthesize protocol mediators, at both the application and middleware layer, in order to achieve interoperability among NSs. We recall that the role of work package WP3 is to devise automated and compositional approaches to CONNECTor synthesis, which can be performed at run-time.

In this deliverable we have presented the results achieved during Year 2 as artifacts of a unified process for the automated synthesis of mediators at both application and middleware layer. All the presented achievements have been applied to a common scenario, i.e., the *Photo Sharing* scenario. In particular, we illustrated the Mediating Connector Architectural Pattern as the key enabler for ensuring interoperability between heterogeneous components. We defined some basic mediator patterns as building blocks for automatic mediator synthesis and sketched out how to effectively deal with them by means of the scenario mentioned above.

We discussed the foundations of CONNECTors, which adapt the protocols run by NSs that implement a matching functionality but possibly mismatch from the standpoint of associated application protocol and even middleware technology used for interactions. We concluded that enabling CONNECTors specifically lies in the appropriate modeling of the NSs high-level functionalities and related protocols, for which we have exploited ontologies so as to enable unambiguous specification. Our main contribution with respect to the state of the art is to deal with both the application and middleware layers. In addition, through the alignment of middleware concepts, we are able to deal with interoperability between networked systems relying on heterogeneous middleware paradigms.

We developed code generation techniques in order to automatically synthesize the actual code that implements the synthesized CONNECTor model. This led us to define also the runtime architecture of CONNECTors by experimenting both connector code generation techniques and generation of code for the run-time interpretation of the connector model.

In order to start to address both functional and non-functional interoperability, we presented a technique to enforce security polices against already synthesized CONNECTors. Furthermore, we started to investigate a combined interoperability approach [16] made by the integration of mediator synthesis with a monitoring mechanism (described in Deliverable D5.2 [7]). The synthesized mediator achieves functional interoperability and the monitor makes it possible to assess the non-functional characteristics of the CONNECTed system at runtime that cannot be assessed statically at synthesis time.

As future work, we intend to define a theoretical compositional strategy to allow reasoning on mismatches and to build the mediating connector behavior. Moreover we also aim at providing the “concrete” basic mediator patterns, i.e., the skeleton code corresponding to the “abstract” ones presented in this deliverable. We plan to implement both the component’s behavior decomposition and the compositional construction of the pattern-based mediator. Data-level mediation as well as algorithms and run-time techniques towards efficient synthesis represent further research directions to be investigated in the future. Our current code generation techniques can still require custom code writing. As future work, we also aim at automatizing these techniques as much as possible. Last but not least, we also need to provide reaction policies or reaction policy patterns that can be undertaken when something wrong is detected by the monitoring. For instance, we could use predictive approaches that try to prevent the wrong behaviors; to adapt the CONNECT architectural infrastructure, if possible, for improving the provided CONNECTion; eventually, to notify the NSs about the unexpected behavior, and let them directly handle the problem.

Bibliography

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

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

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

[4] CONNECT consortium. CONNECT Deliverable D2.2: Compositional algebra of connectors. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[5] CONNECT consortium. 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/.

[6] CONNECT consortium. CONNECT Deliverable D3.2: Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware-Layer, Appendix - Prototype. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[7] CONNECT consortium. CONNECT Deliverable D5.2: Design of approaches for dependability and initial prototypes. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[8] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[9] H. R. Andersen. Partial model checking. In *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 398. IEEE Computer Society, 1995.

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

[11] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[12] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[13] S. Ben Mokhtar, D. Preuveneers, N. Georgantas, V. Issarny, and Y. Berbers. EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. *Journal of Systems and Software*, 81(5):785–808, 2008.

[14] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. In *proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE), Porto, Portugal*, pages 415–429. Springer Verlag, 2005.

[15] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, A. Pathak, R. Spalazzese, B. Steffen, and B. Souville. Towards an architecture for runtime interoperability. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2010.

[16] A. Bertolino, P. Inverardi, V. Issarny, A. Sabetta, and R. Spalazzese. On-the-fly interoperability through automated mediator synthesis and monitoring. In *ISoLA 2010, Part II, LNCS 6416,*, pages 251–262. Springer, Heidelberg, 2010.

[17] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. 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*, pages 141–150, New York, NY, USA, 2009. ACM.

[18] M. Bichier and K.-J. Lin. Service-oriented computing. *Computer*, 39(3):99–101, Mar. 2006.

[19] J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction*. Handbook of Process Algebra. Elsevier, 2001.

[20] Y.-D. Bromberg and V. Issarny. INDISS: interoperable discovery system for networked services. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 164–183, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[21] Y.-D. Bromberg and V. Issarny. Indiss: Interoperable discovery system for networked services. In *Middleware*, pages 164–183, 2005.

[22] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 45–59, London, UK, 1997. Springer-Verlag.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[24] K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communications*, 8(1):127–142, 1990.

[25] T. Cargill. *C++ Programming Style*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[26] L. Cavallaro, E. D. Nitto, and M. Pradella. An automatic approach to enable replacement of conversational services. In *ICSOC/ServiceWave*, 2009.

[27] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[28] F. Chauvel and J.-M. Jézéquel. Code Generation from UML Models with Semantic Variation Points. In L. C. Briand and C. Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3713 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[29] E. Cimpian and A. Mocan. WSMX process mediation based on choreographies. In C. Bussler and A. Haller, editors, *Business Process Management Workshops*, volume 3812, pages 130–143, 2005.

[30] G. Denaro, M. Pezzè, and D. Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of ESEC/FSE 2009*. ACM Press, 2009.

[31] B. P. Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[32] N. Drummond, A. L. Rector, R. Stevens, G. Moulton, M. Horridge, H. Wang, and J. Seidenberg. Putting OWL in order: Patterns for sequences in OWL. In *OWLED*, 2006.

[33] M. Dumas, M. Spork, and K. Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Business Process Management*, pages 65–80, 2006.

[34] E. A. Emerson. Temporal and modal logic. pages 995–1072, 1990.

[35] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[36] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In *ICSE*, pages 771–774, 2006.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[38] P. Grace, G. S. Blair, and S. Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *CoopIS/DOA/ODBASE*, pages 1170–1187, 2003.

[39] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

[40] F. Howar, B. Jonsson, M. Merten, B. Steffen, and S. Cassel. On handling data in automata learning - considerations from the connect perspective. In *ISoLA (2)*, pages 221–235, 2010.

[41] M. Huhns and M. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing Magazine*, 9(1):75–81, Jan. Feb. 2005.

[42] P. Inverardi, V. Issarny, and R. Spalazzese. A theory of mediators for eternal connectors. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2010.

[43] F. Jiang, Y. Fan, and X. Zhang. 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*, pages 3–8, Washington, DC, USA, 2008. IEEE Computer Society.

[44] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[45] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *Knowl. Eng. Rev.*, 18(1):1–31, January 2003.

[46] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: The state of the art. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. (IBFI), Schloss Dagstuhl, Germany.

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

[48] R. Kumar, S. Nelvagal, and S. I. Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems*, 7(3), 1997.

[49] S. S. Lam. Correction to ”protocol conversion”. *IEEE Trans. Software Eng.*, 14(9):1376, 1988.

[50] X. Li, Y. Fan, J. Wang, L. Wang, and F. Jiang. A pattern-based approach to development of service mediators for protocol mediation. In *proceedings of WICSA ’08*, pages 137–146. IEEE Computer Society, 2008.

[51] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[52] F. Martinelli. *Formal Methods for the Analysis of Open Systems with Applications to Security Properties*. PhD thesis, University of Siena, Dec. 1998.

[53] F. Martinelli. Analysis of security protocols as open systems. *Theoretical Computer Science*, 290(1):1057–1106, 2003.

[54] F. Martinelli and I. Matteucci. A framework for automatic generation of security controller. *STVR Journal*, 13 SEP 2010.

[55] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electr. Notes Theor. Comput. Sci.*, 179:31–46, 2007.

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

[57] P. Morris, R. Gray, and R. Filman. Goto Removal Based on Regular Expressions. *Journal of Software Maintenance*, 9(1):47–66, 1997.

[58] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[59] H. R. Motahari Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 731–740, New York, NY, USA, 2010. ACM.

[60] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 3, Washington, DC, USA, 2006. IEEE Computer Society.

[61] O. Nano and A. Zisman. Guest Editors' Introduction: Realizing Service-Centric Software Systems. *IEEE Software*, 24:28–30, 2007.

[62] N. F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, 2004.

[63] N. F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4), 2004.

[64] OASIS Technical Committee. OASIS Web Services Business Process Execution Language v2.0 (WS-BPEL). OASIS Standard, Apr. 2007. available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[65] K. Okumura. A formal protocol conversion method. In *SIGCOMM*, pages 30–37, 1986.

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

[67] C. Ouvans, M. Dumas, A. Ter Hofstede, and W. Van der Aalst. From BPMN Process Models to BPEL Web Services. In *Proceedings of the 5th International Conference on Web Services (ICWS'06)*, pages 285–292, Sept. 2006.

[68] G. Palfinger. State Action Mapper. In *Proceedings of the 4th Conference on Pattern Languages of Programming (PLOP 1997)*, 1997.

[69] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *ISWC*, 2002.

[70] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36:46–52, 2003.

[71] S. Ponnekanti and A. Fox. Interoperability among independently evolving Web services. In *Proc. ACM/IFIP/USENIX Middleware Conference*, pages 331–351, 2004.

[72] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[73] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[74] R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *ECSA*, pages 335–343, 2010.

[75] R. Spalazzese, P. Inverardi, and V. Issarny. 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)*, pages 345–348, 2009.

[76] M. Stollberg, E. Cimpian, A. Mocan, and D. Fensel. A semantic web mediation architecture. In *In Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006*. Springer, 2006.

[77] R. S. Street and E. A. Emerson. An automata theoretic procedure for the propositional μ-calculus. *Information and Computation*, 81(3):249–264, 1989.

[78] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data Knowl. Eng.*, 25(1-2):161–197, 1998.

[79] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[80] R. Vaculín, R. Neruda, and K. P. Sycara. An agent for asymmetric process mediation in open environments. In R. Kowalczyk, M. N. Huhns, M. Klusch, Z. Maamar, and Q. B. Vo, editors, *SOCASE*, volume 5006 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2008.

[81] R. Vaculín and K. Sycara. Towards automatic mediation of OWL-S process models. *Web Services, IEEE International Conference on*, 0:1032–1039, 2007.

[82] J. Van Gurp and J. Bosch. On the Implementation of Finite State Machines. In *Proceedings of the 3rd Annual International Conference on Software Engineering and Applications (IASTED/SEA 99)*. ACTA Press, 1999.

[83] I. Walukiewicz. *A Complete Deductive System for the μ-Calculus*. PhD thesis, Institute of Informatics, Warsaw University, June 1993.

[84] S. K. Williams, S. A. Battle, and J. E. Cuadrado. Protocol mediation for adaptation in semantic web services. In *ESWC*, pages 635–649, 2006.

[85] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[86] F. Zhu, M. W. Mutka, and L. M. Ni. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.