Dynamic connector synthesis: revised prototype implementation

Amel Bennaceur, Luca Cavallaro, Paola Inverardi, Valerie Issarny, Romina Spalazzese, Daniel Sykes, Massimo Tivoli

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D3.3

Dynamic connector synthesis: revised prototype implementation

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

Inria · DOCOMO Euro-Labs · Imperial College · Lancaster University · Thales · Università 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	:	D3.3
Title of Deliverable	:	Dynamic connector synthesis: revised prototype implementation
Nature of Deliverable	:	Report, Prototype
Dissemination Level	:	Public
Internal Version Number	:	0.1
Contractual Delivery Date	:	31 January 2012
Actual Delivery Date	:	21 February 2012
Contributing WPs	:	WP3
Editor(s)	:	Massimo Tivoli (UNIVAQ)
Author(s)	:	Amel Bennaceur (Inria), Luca Cavallaro (Inria), Paola Inverardi (UNIVAQ), Valérie Issarny (Inria), Romina Spalazzese (UNIVAQ), Daniel Sykes (Inria), Massimo Tivoli (UNIVAQ)
Reviewer(s)	:	Nikolaos Georgantas (Inria)

Abstract

The CONNECT *Integrated Project aims at enabling continuous composition of Networked Systems (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 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 efficient 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 second year, the work of WP3 has been mainly focused on defining a unified process, and related artifacts, for the automated synthesis of mediators at both application and middleware layers. The devised mediator synthesis process allows for the automated production of both a mediator behavioral model and its implementation into actual code. All this work relies on a CONNECT*or theory that rigorously characterizes concepts, such as protocol abstraction, matching, and mapping, that are crucial for achieving automated mediator synthesis. The theory, as formalized during the first two years of the project, is able to deal with an abstract notion of protocol action that did not take into account data as I/O action parameters.*

During the project's third year, as described in this deliverable, we enhance our CONNECT*or theory in order to consider actions with data and rigorously characterize the abstraction and matching phases that have been treated informally by the previous theory. Furthermore we propose two possible instantiations of the theory, both of them implemented by two prototype tools. The first instantiation accounts for a specification of the goal that the* CONNECT*or to be synthesized has to achieve. The second instantiation concerns the ability to perform efficient protocol mapping.*

Considering the goal allows the synthesis approach to harmonize the interaction of two NSs by only looking for some possible matchings. However, when such a goal is not provided, this approach exhaustively explore the whole NSs interaction. The mapping-driven synthesis, instead, is well suited for the cases where the goal is not specified. Moreover, it checks the existence of the mediator by avoiding to perform expensive synthesis steps when a mediator for two NSs does not exist. Furthermore, there are protocol mismatches that can be handled more effectively with the goal-based approach and mismatches for which the mapping-driven method represents the best choice among the two approaches. The developed prototypes and their respective CONNECT*or synthesis methods have been validated by means of a common case study. Based on the experimented results, an assessment and comparison of the two methods have been carried out.*

Keyword List

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

Table of Contents

List of Acronyms

Acronym	Meaning
CP	Constraint programming
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
eLTS	extended Labeled Transition System
FSP	Finite State Processes
LTS	Labeled Transition System
eLTS	enhanced Labeled Transition Systems
LTSA	Labeled Transition System Analyzer
NS(s)	Networked System(s)
SWS	Semantic Web Service

List of Notations

Notation	Meaning
$c \sqsubseteq d$	Ontology subsumption - the concept c is subsumed by the concept d
$\overline{act}$	Output action - dual action of the input action act
αP	Process alphabet - the set of action a process can engage in
$s_i \xrightarrow{l} s_j$	A process transition from state s_i to state s_j after engaging in an action l
$s \stackrel{l}{\Rightarrow} s'$	A shorthand for $s \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 ... \xrightarrow{l_n} s', l = \langle l_1, l_2, ..., l_n \rangle, l_i \in \alpha P \cup \tau$
$X \mapsto Y$	X maps to Y - there is a correspondence between the action (or sequence of actions) X and the action (or sequence of actions) Y
$P \leftrightarrow_M Q$	P and Q are behaviorally compatible through a mediator M

List of Figures

List of Tables

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of Networked Systems (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 efficient approaches to CONNECTor synthesis, which can be performed at runtime. 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.

A high level view of the CONNECT architecture is described, in Section 5.1 of Deliverable D1.1 [2], as a system of various enablers that exchanges information about the NSs to be CONNECTed. In particular, as described in details in Chapter 6 of this deliverable, the *Synthesis* enabler takes as input: (i) a description of two NSs, which contains the information about the NS interface and protocol, and (ii) a, possibly empty, list of goals that the CONNECTor to be synthesized has to achieve. From these two inputs, the Synthesis Enabler produces a CONNECTor behavioral model that is interpreted at run-time [4] and hence executed in order to make the two considered NSs able to interoperate.

During the project's second year, the work of WP3 has been mainly focused on defining a unified process, and related artefacts, for the automated synthesis of mediators at both application and middleware layers. The devised mediator synthesis process allows for the automated production of both a mediator behavioral model and its implementation into actual code. In fact, a further outcome of the project's second year has been the development of suitable code-generation techniques to generate the actual code that implements a synthesized CONNECTor. All this work relies on a CONNECTor theory that rigorously characterizes concepts, such as protocol abstraction, matching, and mapping, that are crucial for achieving automated mediator synthesis. The theory, as formalized during the first two years of the project, is able to deal with an abstract notion of protocol action that did not take into account data as I/O action parameters.

During the project's third year, as described in this deliverable, we:

- Enhance our CONNECTor theory in order to both account for actions with data and describe in more detail the abstraction and matching phases that have been discussed more abstractly in the previous version of the theory.

- Propose two possible instantiations of the theory, both of them implemented by two prototype tools [10]. The first instantiation accounts for a specification of the goal that the CONNECTor to be synthesized has to achieve. The second instantiation concerns the ability to perform efficient protocol mapping through the combination of semantic reasoning and constraint programming techniques.

Considering the goal allows the synthesis approach to mediate only partial matching between the behavior of the considered NSs. However, when such a goal is not provided, this approach exhaustively explores the whole matching. Therefore, the mapping-driven synthesis is well suited for the cases where the goal is not specified as it considers the whole behavior of systems and checks that each possible execution of one system can possibly be mapped to an execution of the other system. Moreover, the mapping-driven synthesis checks the existence of the mediator by avoiding to perform expensive synthesis steps when a mediator for two NSs does not exist. However, many-to-many mismatches with asynchronous semantics cannot be handled effectively by the mapping-driven synthesis. The goal-based approach, instead, bases its matching phase on a reachability problem hence accounting for both synchronous and asynchronous behaviors and loops. On the other hand, the goal-based approach is able to produce only subsets of the whole CONNECTor in those cases in which the interactions allowed by the two NS protocols are infinite.

The developed prototypes and their respective CONNECTor synthesis methods have been validated by means of a common case study that is introduced in Section 1.5. The two developed implementations

has led us to devise a first concrete architecture for the CONNECTor synthesis enabler, which is one of the key elements of the CONNECT architecture.

Finally, it is worthwhile mentioning that an overall challenge of the work carried out within WP3 is to prevent human intervention as much as possible by making the CONNECTor generation process completely automated. The work mentioned above and described in this deliverable also addresses this overall challenge.

This deliverable is organized as follows. In Chapter 2, we formalize the enhanced CONNECTor theory. Chapters 3 and 4 describe the two instantiations of the revised theory that have been mentioned above. The goal-based instantiation is described in Chapter 3 while the mapping-driven synthesis is discussed in Chapter 4. Chapter 5 assesses the two instantiations with respect to their ability to address a common set of protocol mismatches hence providing a comparison of them, and discusses related work. The Synthesis Enabler architecture and its related API is described in Chapter 6. Chapter 7 concludes by also outlining future research directions.

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 runtime. 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 runtime, 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 runtime, 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.

We recall that WP3 is organized into three tasks that correspond to the three objectives above. Section 1.4 discusses what we did during the Year 3 of the project within the context of each specific task.

1.2 Brief Summary of Achievements in Year 2

During the second year of the project, the work of WP3 mainly focused on defining a comprehensive CONNECTor synthesis process together with supporting methods and tools, based on the theory of mediators introduced in Deliverable D3.1 [7]. In particular, our previous work has led us to develop automated techniques and tools to perform protocol abstraction, matching, and mapping, and code-generation techniques to derive from the synthesized CONNECTor model its actual code.

In more detail, the challenges that have been addressed during Year 2, and the related achievements, include the following, as detailed in Deliverable D3.2 [9]:

- **Approaching the CONNECTor synthesis problem in a systematic way by adopting a pattern-based solution.** 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.

- **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.

- **From theory to abstract CONNECTor synthesis.** While our theory of mediators was defined over protocols defined in terms of highly abstract observable actions, actual CONNECTor synthesis required 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 had to account for the semantics of actions at both application- and middleware-layers.

- **From abstract to concrete CONNECTor deployment.** Translating the synthesized CONNECTor model into an executable artifact that could be deployed and run required devising the runtime architecture of CONNECTors; related to this, in Year 2 we investigated the issue of generation of code versus interpretation of the CONNECTor model.

The software tools developed during Year 2 as companion prototypes of Deliverable D3.2 are reported in [8].

1.3 Second Review Recommendations and Related Reactions for Year 3

The following list of items report the reviewers' recommendations for the work done within WP3 after the second year of the project. For each recommendation, the reactions of the WP3 participants are also reported.

- *Recommendation 1: the reviewers still query how the intent to connect is expressed. In particular, while affordances specify provisions to and requirements from an environment they do not express the goal of connection. For what purpose does a set of entities interconnect? Where are these goals and intents expressed in an overall CONNECT architecture consisting of agents/components and connectors?*

  We believe that this specific question should not be posed to WP3 only, rather it is an issue that impacts on the overall CONNECT architecture and it is overall addressed by WP1. However, within the WP3 work, both our enhanced theory and its two described instantiations have a notion of goal that can be either explicitly expressed by using a notation that is convenient for the particular synthesis method or implicitly treated for the purposes of mediator synthesis. In particular the notion of *coordination policy*, as defined by our enhanced theory, represents a *relevant behavior* exhibited by a NS, in the sense that it corresponds to the achievement of a complex task. Thus, within our theory, the goal that the mediator has to satisfy is to make the two considered NSs able to interoperate on, at least, one common relevant behavior. This abstract notion of goal is instantiated by the goal-based synthesis method by considering the goal explicitly specified as a formula in some temporal logic formalism. Then, analogously to what the theory formalizes, it is sufficient to synthesize a mediator that make the considered NSs able to interoperate just with respect to the interactions expressed by the goal's specification. Instead, within our mapping-driven instantiation of the theory, no explicit goal specification is considered. However, for a synthesized mediator to be correct, it is enough to match each possible behavior exhibited by one NS with, at least, one behavior of another NS.

- *Recommendation 2: also WP3 proposes a framework for compositional reasoning. The compatibility between the WP2 and the WP3 settings is however unclear. Also, the approach sketched in WP3 (section 6 of the deliverable) looks rather naïve, and not much is proposed to avoid combinatorial explosion.*

  Our method to automatically secure connectors is based on partial model checking techniques that have been just defined with the aim of avoid combinatorial explosion in mind.

1.4 Challenges for Year 3 and Overview of the Related Achievements

During Year 3, the main overall challenge has been to concretely realize and automate the unified process, devised during Year 2, for the synthesis of mediators at both application and middleware layer. The specific per-objective (and, hence, per-task) achievements include the following. This deliverable describes in details only some of them and, in particular, the ones that at present are mature enough to be discussed in a project deliverable document.

- *Objective/Task 1* and *Objective/Task 2:*
  - revised version of the theory that both allows to consider protocol actions with data and rigorously characterizes the protocol abstraction and matching phases of the synthesis process (described in this deliverable).
  - Goal-based synthesis method as first instantiation of the revised theory; it allows to address signature mismatches, splitting of actions, extra-output mismatches, and, only for asynchronous systems, merging and ordering of actions; it produces a complete CONNECTor only if the considered systems do not exhibit infinite looping behavior (described in this deliverable).
  - Mapping-driven synthesis method as second instantiation of the revised theory; it allows to address signature mismatches, splitting of actions, extra-output mismatches, and, only for input actions that do not require output paramenters, merging of actions; it always produces a complete CONNECTor (described in this deliverable).
  - Two prototypal tools implementing the above goal-based and mapping-driven synthesis methods, respectively. We refer to Deliverable D3.3 - Protoype Appendix [10] for details concerning these two tools.
  - A theory for the automatic construction of secure connectors able to mediate and adapt networked systems that have cryptographic primitives for exchanging messages. Basically, given two networked systems, described through LTSs with enriched semantics for managing symbolically cryptographic functions as encryption, decryption, hash, etc. and a security contract expressing how messages should be correctly exchanged among those, it synthesizes a secure connector, able to enforce the correct exchange of cryptographic messages and able to resist to the presence of hostile attackers (not described in this deliverable).
- *Objective/Task 3:* we implemented two model-driven synthesis tools that implement the goal-based and mapping-driven synthesis methods listed above.

1.5 Case Study

The GMES scenario demonstrates the applicability of the CONNECT approach in real-world settings and highlights the integration of the different CONNECT enablers. In this deliverable we focus on the synthesis of mediators and illustrate its benefits in dealing with a wide range of mismatches while comparing it with existing approaches. Therefore, we used the Purchase Order Mediation scenario from Semantic Web Service (SWS) Challenge[1]. It represents a typical real-world problem that is as close to industrial reality as practical. It is intended as common ground to discuss semantic (and other) Web Service solutions and make different solutions becoming comparable with respect to the set of features supported for a particular scenario. This scenario highlights the various mismatches that can be encountered when making heterogeneous systems interoperable. The Purchase Order Mediation scenario describes two commercial systems that have implemented using heterogeneous industrial standards and protocols.

The first system, called *Blue*, is a customer ordering products. It initiates a purchase process by starting an order and adding items to it. Then, it places the order giving its client identifier. Finally, it expects a confirmation for each individual item belonging to the order. The exchanged information is

[1] http://sws-challenge.org/wiki/

formatted using the RosettaNet[2]. RosettaNet is an XML-based standards for the global supply chain and interaction across companies.

The second system, called *Moon* uses two backend systems to manage its order processing, namely a Customer Relationship Management system (CRM) and an Order Management System (OMS). First, a client contacts the Customer Relationship Management (CRM) System to obtain relevant customer details. This details are used by the OMS to assess if the client is eligible, i.e. if the customer is known and authorized to creating order. Then, individual items can be added to the order created. First an item is selected, the needed quantity is specified and the the addition to the order is confirmed. Once all the items have been submitted, the Moon proceed to payment using a third-party system and the order can be closed.

A client developed for the Blue Service cannot communicate with the Moon Service due to the following mismatches:

- *Data mismatches:* While Blue specifies its interface using the RosettaNet standard, Moon uses a propriety legacy system in which data model and message exchange patterns differ from those of RosettaNet.
- *Behavioral mismatches:* In the Blue implementation, the client provides its identifier while placing the order whereas in the Moon implementation it has to login before performing any operation. In addition, in the Blue implementation, an item is directly added and only once the order is placed then confirmations are sent while in Moon, first the item is selected, the quantity is specified and then confirmed. Hence, there is no need to send confirmations once the order has been closed.

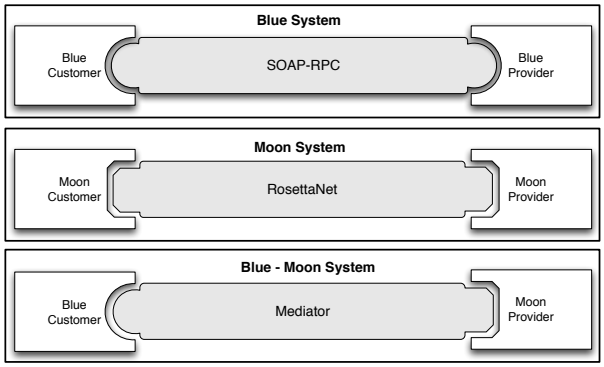

Figure 1.1: The purchase order mediation scenario

The SWS-Challenge provides relevant information about the systems involved in two forms: using current Web Service description (WSDL) and natural language text annotations. We interpreted the information, defines the ontology and annotated the description. Indeed, the SWS-Challenge participants are asked to extend the syntactic descriptions in a way that their algorithm/systems can perform the necessary translation tasks in a fully automatic manner. The Moon and the customer Web Services (Blue) are provided by the SWS-Challenge organizers and can not be altered (although their description may be semantically enriched).

[2] http://www.rosettanet.org/

2 Revised CONNECTor Theory

In this chapter we recall the unified CONNECTor Synthesis approach presented last year in Deliverable D3.2 [9] that deals with both middleware and application layers and we provide a revised version of the Theory supporting it.

2.1 Recalling Second Year Achievements

Figure 2.1 outlines our overall approach to support the dynamic synthesis of mediators given Networked Systems (NSs) models and the ontologies describing the domain-specific knowledge.

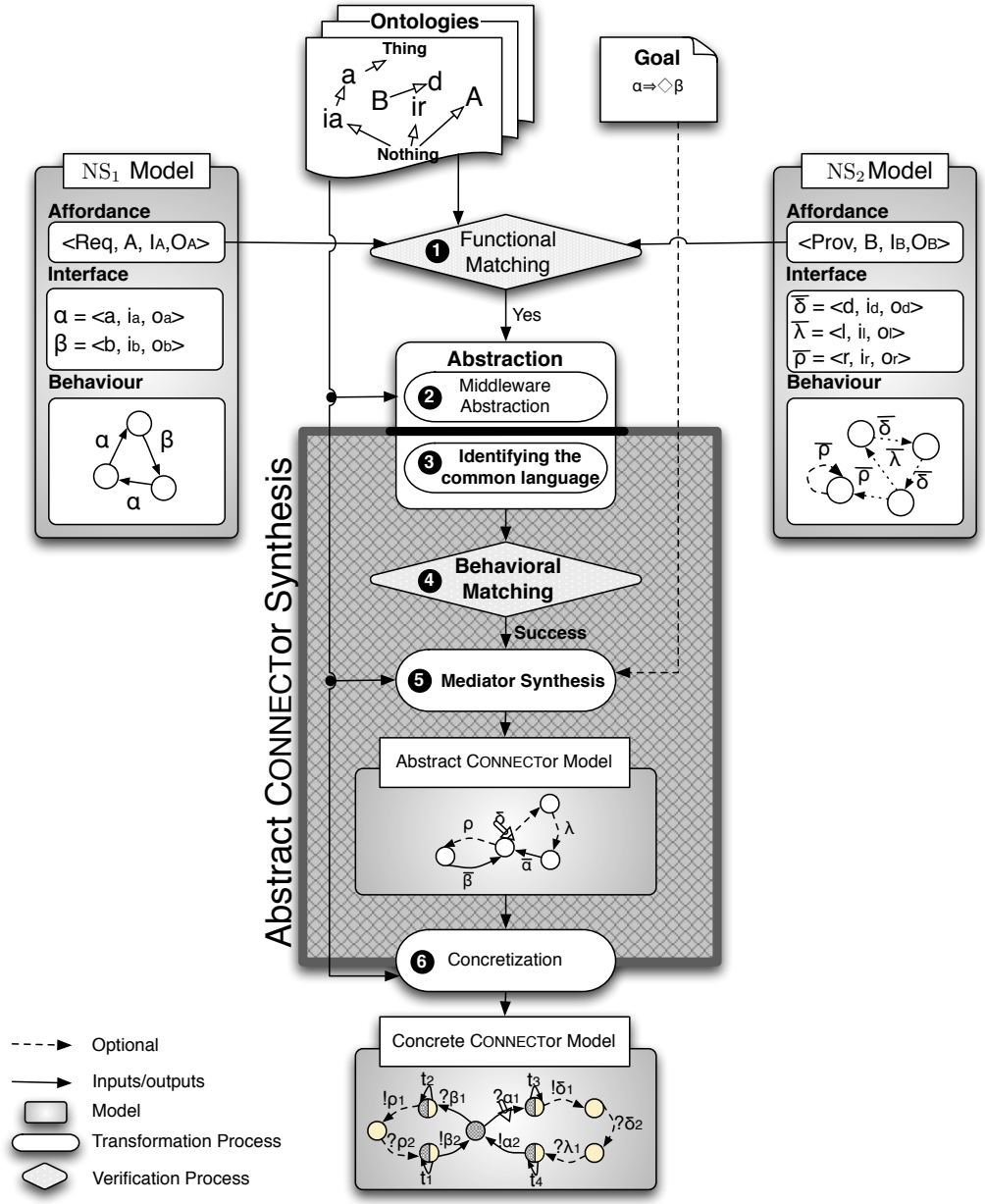

Figure 2.1: Overview of the CONNECTor synthesis

The Abstract CONNECTor Synthesis, as presented last year, is preceded by *Functional Matching* and

Middleware Abstraction. The Functional Matching consists of checking whether, at a high level of abstraction, the functionality required by one system can be provided by the other (see Figure 2.1 ❶). This is realized by checking the semantic compatibility of the NSs affordances using an ontology reasoning. Instead, the Middleware Abstraction makes the behavior of NSs middleware-agnostic , i.e., it abstracts/-translates (sequences of) middleware functions into input/output direction of application actions in order to focus on the application-specific semantics (see Figure2.1 ❷).

The Abstract CONNECTor Synthesis process is made up by three phases or steps: *Identification of the Common Language* makes comparable the NSs behavior by identifying their common language and, possibly, reduces their size thus allowing to ease and speed up the reasoning on them (see Figure2.1 ❷). The next step is the *Behavioral Matching* (see Figure2.1 ❹) that checks the NSs compatibility identifying possible mismatches. Finally, the *Mediator Synthesis* (see Figure2.1 ❺) produces a mediator that address the identified mismatches between the two NSs and allows them to communicate.

After the Abstract CONNECTor Synthesis, the synthesized mediator is concretized to be deployed in the network in order to enable the NSs to interoperate.

In this deliverable we focus on the Abstract CONNECTor Synthesis (❸, ❹, ❺). For the middleware abstraction, we rely on the approach devised in deliverable D3.2 [9] while the dual concretization process is described in deliverable D1.3 [4]. The aforementioned conceptual view is supported by (a) a revised CONNECTor theory for the application layer in Section 2.2, together with (b) two different prototypal implementations of it illustrated in Chapters 3 and 4 respectively.

2.1.1 Ontologies

An ontology is *a shared, descriptive, structural model, representing reality by a set of concepts, their interrelations, and constraints under the open-world assumption* [12].

In the literature, [49, 48], ontologies and mappings between ontologies are formalized as follows:

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

Ontologies describe domains; therefore, they are not defined by the application developers but by domain experts, to represent shared knowledge about a specific domain. The Web Ontology Language[1] (OWL) is a W3C standard language to formally model ontologies in the Semantic Web. Many OWL ontologies have been developed for specific domains, e.g., Sinica BOW[2] (Bilingual Ontological Wordnet) for English-Chinese integration. In addition, work on ontology alignment deals with the possible usage of distinct ontologies in the modeling of different networked systems from the same domain, as illustrated by the W3C Linking Open Data project[3]. OWL is based on description logics (DL), which is a knowledge representation formalism with well-understood formal properties [13]. Concepts are defined as OWL classes. Complex concepts can be constructed using intersection ($\cap$), union ($\cup$) and complement ($\neg$). Instances of a class are called OWL individuals and can be automatically classified in the corresponding class. Relations between classes are called OWL properties. Ontology reasoners are used to support automatic inference on concepts in order to reveal new relations that may not have been recognized by the ontology designers. Traditionally, the basic reasoning mechanism is based on the subsumption relation that is defined as follows [13].

Definition 1 (Subsumption) *A concept c is subsumed by a concept d in a given ontology $\mathcal{O}$, written $c \sqsubseteq d$, if in every model of $\mathcal{O}$ the set denoted by c is a subset of the set denoted by d.*

[1] http://www.w3.org/TR/owl2-overview/
[2] http://BOW.sinica.edu.tw/
[3] http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

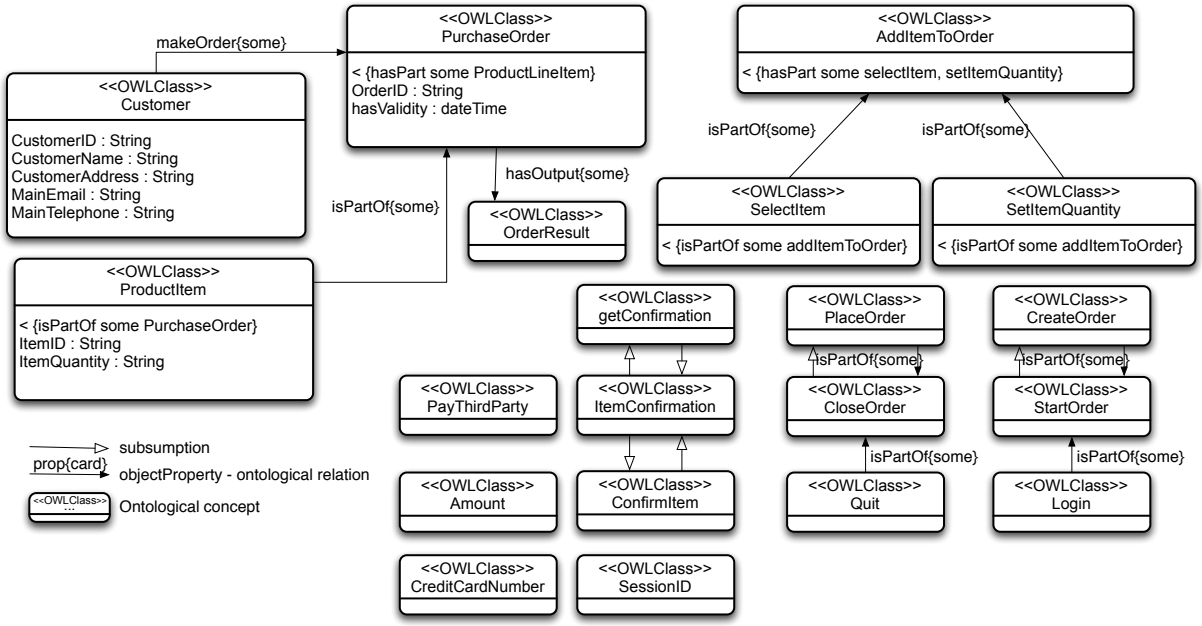

Figure 2.2: The purchase ontology

Further, other relations may be defined between ontology concepts, e.g., part-whole relationship, and the subsumption relation can be used to implement other inferences, such as satisfiability and equivalence, using pre-defined reductions.

Figure 2.2 shows an extract of the purchase ontology. The ontology shows the relations holding between the various concepts used by two purchase order systems. It specifies the attributes of each concept; for example a *PurchaseOrder* is characterized using two properties: *hasOrderID* defined as an OWL string and *hasValidity* specified using the OWL buit-in dateTime type. Note that the application-specific ontology not only describes the semantics and relationships related to the data but also the semantics of the operations performed on the data. The operation *addItemToOrder* is for example defined as a union of the *selectItem* and *setItemQuantity* concepts.

2.1.2 Ontology-based Networked System Model

A networked system requires or provides an *affordance* to which it gives access via an explicit *interface*, and realizes using a specific *behavior* [9].

The *affordance* describes the functionality encapsulated by a system and is specified as a tuple: $\mathcal{F}$=<*type*, c, in, out> where (i) *Type* stands for provided (noted *Prov*) if the system is offering this functionality or required (noted *Req*) if it is consuming it, (ii) c gives the semantics of the functionality in terms of an ontology concept; (iii) in (out resp.) specifies the set of inputs (outputs resp.) of the functionality, which is defined as a set of ontology concepts.

All concepts belong to the same domain ontology $\mathcal{O}$ specifying the application-specific concepts and relations, i.e., $c, in, out \in \mathcal{O}$. As an illustration, the affordance of Moon service is defined as: $\mathcal{F}_{Moon}$=<*Prov, Order,* $\emptyset$, $\emptyset$>. while that of the Blue client is defined as: $\mathcal{F}_{Blue}$=<*Req, Order,* $\emptyset$, $\emptyset$>.

The system *interface* defines the set of observable *actions* that the system requires/provides from its running environment. An observable action is defined more formally as follows.

Definition 2 (*Action*) *An action is a tuple* $<op,In,Out>$ *where:*

- op *is an observable operation referring to an ontology concept or is an internal action denoted by* τ*; an action can have input/output direction denoted by an overbar or no overbar on the action, e.g.* $\overline{act}$

*or act. We call **complementary actions** two actions that are the same action with dual direction, e.g., $\overline{act}$ and act.*

- *In, Out are the sets of input/output data that can be expected/produced whose elements refer to ontology elements. One at a time of these sets might also be empty either because that dataset is not necessary or because the synchronization of the input output is not atomic.*

Consequently, we are able to describe the following actions with data:

1. *output action with incoming parameters and outgoing return data:* $<\overline{op}, In, Out>$ *where In is expected while Out is produced;*
2. *input action with outgoing parameters and incoming return data:* $<op, In, Out>$ *where In is produced while Out is expected;*
3. *output action with incoming parameters and no return data:* $<\overline{op}, In, \emptyset>$ *where In is expected -and no Out is produced;*
4. *input action with incoming parameters and no return data:* $<op, In\ \emptyset>$ *where In is produced -and no Out is expected;*
5. *output action with outgoing return data and no incoming parameters:* $<\overline{op}, \emptyset, Out>$ *where Out is produced -and no In is expected;*
6. *input action with incoming return data and no outgoing parameters:* $<op, \emptyset, Out>$ *where $Out$is expected -and no In is produced;*

Note that the first kind of action illustrated above, $<\overline{op},In,Out>$, can be equivalently described by the two following action primitives: $<\overline{op},In,->$ and $<\overline{op},-,Out>$. This applies similarly for the second kind of action described above, i.e., $<op, In, Out>$.

The interface associated to the Moon service (abstracted from WSDL 2.0) is given below, where we provide only the ontology concepts associated with the syntactic terms embedded in the interface.

$\mathcal{I}_{Moon}$ ={
<Login, {CustomerID}, {SessionID}>,
<$\overline{\text{CreateOrder}}$, {SessionID}, {PurchaseOrder}>,
<$\overline{\text{SelectItem}}$, {OrderID, ItemID}, $\emptyset$>,
<$\overline{\text{SetItemQuantity}}$, {ItemID, ItemQuantity}, $\emptyset$>,
<ConfirmItem, {OrderID, ItemID, ItemQuantity}, $\emptyset$>,
<$\overline{\text{CloseOrder}}$, {OrderID, CreditCardNumber}, {OrderResult}>,
<PayThirdParty, {CreditCardNumber, Amount}, {OrderResult}>
}

The interface associated to the Blue client is shown in the following.

$\mathcal{I}_{Blue}$ ={
<StartOrder,{CustomerID}, {OrderID}>,
<AddItemToOrder,{OrderID, ProductItem}, $\emptyset$>,
<PlaceOrder,{OrderID, CreditCardNumber}, $\emptyset$>,
<Quit,{OrderID},{OrderResult}>
<$\overline{\text{GetConfirmation}}$,{OrderID, ProductItem}, $\emptyset$>,
}

The system *behavior* describes its interaction with its environment and defines how the actions of its interface are used to achieve the specified affordance. We formalize the behavior through enhanced Labeled Transition Systems that are Labeled Transition Systems [50] enhanced with explicit final states and which labels are structured to explicitly model input/output actions. More formally:

Definition 3 (*enhanced Labeled Transition Systems*) *An enhanced Labeled Transition Systems (eLTS)* P *is a quintuple* (S, L, D, F, s_0) *where:*

- S *is a finite non-empty set of states;*
- L *is a finite set of labels describing actions with data (see Definition 2); L is called the alphabet of* P*:*
- $D \subseteq S \times L \times S$ *is a transition relation;*
- $F \subseteq S$ *is the set of final states;*
- $s_0 \in S$ *is the initial state;*

We use the usual following notation to denote transitions: $s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$

Examples of eLTSs are the Blue application protocol and the Moon application protocol of our case study that we copy here in Figures 2.3 and 2.4 respectively.

LTSs, and eLTSs, can be combined using the parallel composition operator. Several semantics have been given in the literature for this operator, e.g. for CSP (*Communicating Sequential Processes*) [66], or FSP (*Finite State Process*) [55], or also CCS *Calculus of Communicating Systems* [59] to mention few. The one needed here is the one of CCS, recalled below, that applies *after having aligned the eLTSs alphabets into the same alphabet and having a language restricted to the mediator language*.

Definition 4 (*Parallel composition of protocols*) *Let* $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$ *be eLTSs. Let* m *and* $\overline{m}$ *be actions.*
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:*

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

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

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

2.1.3 Background of the Mapping-driven Abstract CONNECTor Synthesis

We recall that last year we presented a first prototype of the Mapping-driven Abstract CONNECTor Synthesis where the behavior of Networked Systems is given as Ontology-based Finite State Process (OFSP) which gives formal semantics to the actions of an FSP process [55] by relating them to the ontology-based specification of the interface $\alpha P = \mathcal{I}_P$. The semantics of OFSP is given in terms of enhanced Labeled Transition Systems (eLTS). The interface $a \in \mathcal{I}_P$ represents the behavior of the eLTS P after it engages in an action a. $P \xrightarrow{a} P'$ then denotes that P transits with action a into P'. Then, $P \overset{s}{\Rightarrow} P'$ is a shorthand for $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 ... \xrightarrow{a_n} P', s = \langle a_1, a_2, ..., a_n \rangle, a_i \in \alpha P \cup \tau$. The END state indicates a successful termination.

The OFSP specification of the Blue client is as follows:

```
1  Blue  =  (<StartOrder, {CustomerID}, {OrderID}>→ P1),
2  P1    =  (<AddItemToOrder,{OrderID, ProductItem}, ∅>→ P1
3           | <PlaceOrder,{OrderID, CreditCardNumber}, ∅>→ P2),
4  P2    =  (< GetConfirmItem,{orderID, productItem}, ∅>→ P2| <Quit,{OrderID},{OrderResult}>→ terminate→ END).
```

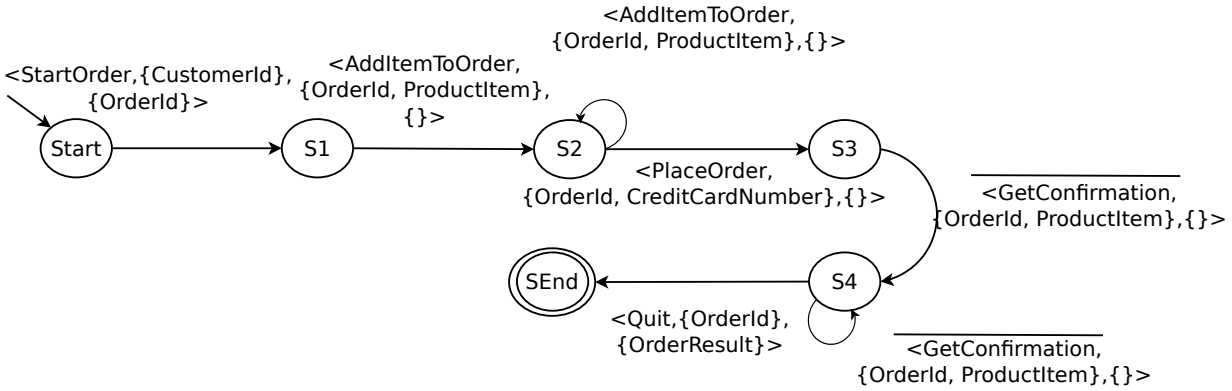

Figure 2.3: *Blue* client networked system.

The corresponding eLTSs is depicted in Figure 2.3. Blue starts an order and then adds one or many items to it. Once the order is placed, the confirmation for each individual item is received.

The Moon OFSP specification is as follows:

```
1  Moon = (<Login, {CustomerID}, {SessionID}>→<CreateOrder, {SessionID}, {PurchaseOrder}>→ P1),
2  P1   = (<SelectItem, {OrderID, ItemID}, ∅>→<SetItemQuantity, {ItemID, Quantity}, ∅>
3          →<ConfirmItem, {OrderID, ItemID, Quantity}, ∅>→ P2
4          | <CloseOrder, {OrderID, CreditCardNumber}, ∅>
5          →<PayThirdParty, {CreditCardNumber, Amount}, {OrderResult}>
6          →<CloseOrder, ∅, {OrderResult}>→ terminate→ END).
```

Its semantics is given if the eLTS depicted in Figure 2.4. Moon, instead, expects a client to login, and to create an order. Then it can loop on selecting an item, setting the quantity of that item and confirming it. Finally, it closes, pays the order and terminates.

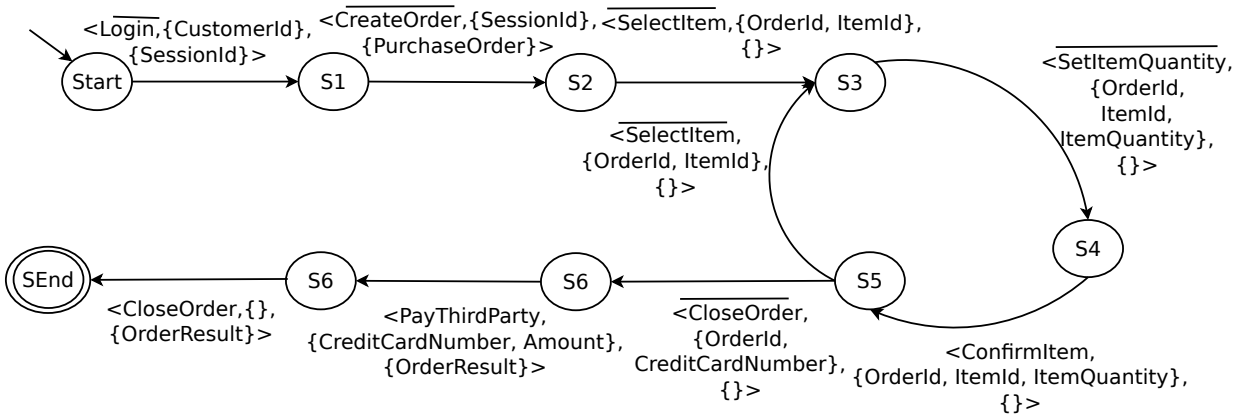

Figure 2.4: *Moon* customer service networked system.

2.2 Updated CONNECTor Theory

In this section we present a refined version of the CONNECTor Theory to synthesize a CONNECTor between two protocols presented in the last year deliverable and in [43, 69, 42, 72]. The novelties of the theory described in the reminder of this chapter with respect to the previous version are:

1 - in the theory below, together with observable application actions, we also explicitly model the data that the actions convey. Instead, in the previous formalization presented in Deliverable D3.2 [9], we abstracted the NSs application behavior as LTSs expressing the order in which input and output (observable) application actions have to be performed while interacting with the NS. Specifically, input actions were used to model (i) methods that can be called, or (ii) the end of receiving messages from communication channels, as well as (iii) the return values from such calls. While output actions were used to model (i) method calls, (ii) message transmission via communication channels, or exceptions that occur during methods execution.

2 - In the theory presented below, we generalize the manipulations we need to do all along the abstraction (identification of the common language) and behavioral matching phases that are the costly and hard ones while in the previous version they were more implementation specific.

We recall that our focus is the interoperability problem between heterogeneous protocols. For the sake of simplicity, and without loss of generality, we limit the number of protocols to two but the work can be generalized to an arbitrary number of protocols. With **interoperability**, we mean the ability of protocols to *correctly communicate and coordinate* i.e., to *correctly synchronize*. In other words, two systems successfully interoperate if they correctly exchange *compatible conversations* or *compatible traces*. The kind of protocols we focus on is **compatible protocols** i.e., that can *potentially interoperate* despite they show some differences. That is, communication and coordination between such protocols is possible in principle since they are semantically equivalent and complementary, but cannot be achieved seamlessly because of *heterogeneity* or *diversities*: *mismatches* [71, 70] and/or *third parties conversations*. Examples of mismatches are: protocol languages have (i) different granularity, or (ii) different alphabets; protocols behavior have different sequences of actions with data (i.e., traces) because of (a.1) the order in which actions and data are performed by a protocol is different from the order in which the other protocol performs the complementary actions with data. Protocols behavior may have different sequences of actions also because of (a.2) interleaved actions related to *third parties conversations* i.e., with other systems, the environment. In some cases, as for example (i), (ii) and (a.1), it is necessary to properly perform a manipulation of the two languages. In the case (a.2) it is necessary to abstract the third parties conversations that are not relevant to the communication. *Synchronization* between protocols, thus, can be achieved *under mediation* i.e., through a mediator that while managing mismatches and third parties conversations, allows protocols to effectively exchange compatible traces (sequences of actions with data). A **mediator** is then a protocol that allows communication and coordination among compatible protocols by mediating their differences. Therefore, such a mediator serves as the locus where *semantically equivalent and complementary actions with data* are correctly synchronized thus enabling (a mediated) interoperability among protocols.

We already proposed to automatically synthesize CONNECTors also called *mediating connectors* or *mediators*, and we provided a theory to characterize and reason on the problem [43, 42, 72, 69]. In the remainder of this chapter we revise our *theory of mediators*. By reasoning about the mismatches of the compatible protocols, our theory automatically identify and synthesizes an *emerging mediator* that solves them thus allowing protocols to interoperate.

To better illustrate the CONNECTor theory, in this chapter we will use the case study presented in Section 1.5 that is constituted by applications Blue client networked system (Blue) and Moon customer service networked system (Moon) with compatible protocols i.e., semantically equivalent and complementary, that in principle might interoperate but that in practice cannot communicate and coordinate seamlessly because of mismatches.

2.2.1 Detailing the Automated Synthesis of Emerging Mediators

This section provides a more detailed overview of our approach towards the automated synthesis of emerging mediators. Figure 2.5 depicts the main elements.

1. Two application-layer protocols P and Q (e.g., Blue and Moon) whose behavioral representations are given in terms of enhanced Labeled Transition System (eLTS), where the *initial* and *final states* on the eLTSs define the *sequences of actions* that characterize the *coordination policies*(or *traces*) of the protocols and where labels explicitly models the data conveyed by the application actions.

2. The *application domain ontology* O and two *sub-ontologies* of it, i.e., $O_P \subseteq O$ and $O_Q \subseteq O$, describing the meaning of P and Q's actions and input/output data, respectively (e.g., O_P is Blue's

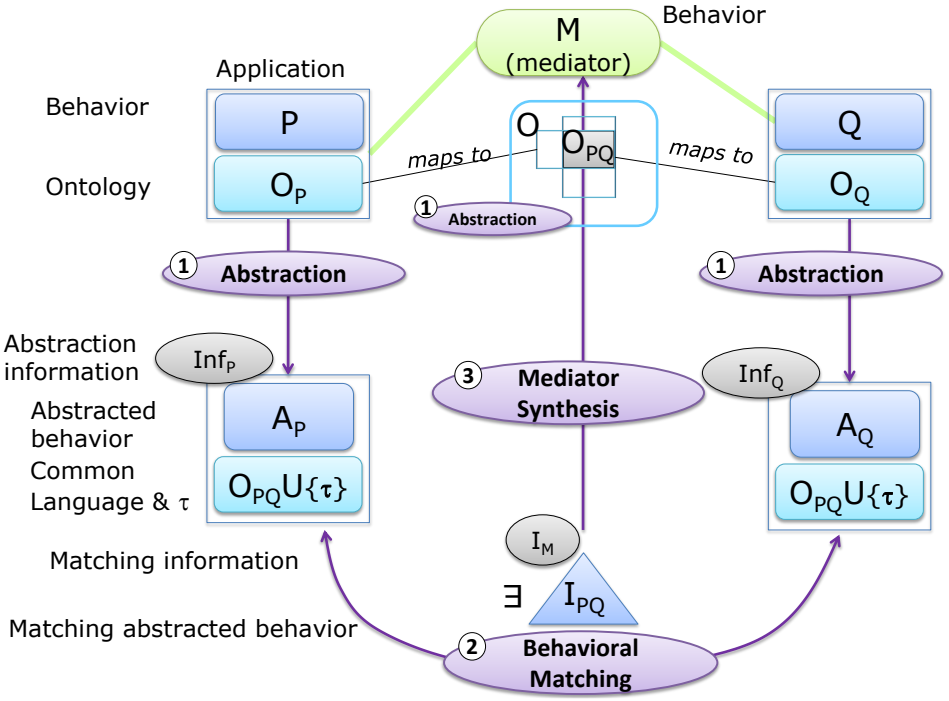

Figure 2.5: An overview of our approach

ontology and O_Q is Moon's ontology). The inferred *emerging common abstraction* O_{PQ} is a kind of intersection between P and Q's ontologies ($O_P \bigcap O_Q$) and identifies the *common language* between P and Q.

3. Then, starting from P and Q, and based on the common language O_{PQ}, we build two abstractions A_P and A_Q of P and Q, respectively, where the actions not belonging to the common language are hidden by means of silent actions (τs). We store the abstraction information Inf_P and Inf_Q, which include ontological mappings for instance solving granularity mismatches[4]. This information will be exploited to synthesize the mediator in case of successful behavioral matching;

4. Then, we check the *behavioral matching* or compatibility of the protocols by looking for *complementary traces* (the set I_{PQ} in figure), modulo mismatches and third parties communications, between the sets of traces of A_P and A_Q, respectively. If protocols are compatible, then we are able to synthesize a mediator that makes it possible for the protocols to communicate and we store the matching information I_M (i.e., used to make the behavioral matching) that will be exploited during the mediator synthesis. Notice that with synchronization we mean a synchronization among two complementary actions while for coordination we mean a sequence or a reordering of synchronizations needed to synchronize complementary traces.

5. Finally, given two protocols P and Q, and an environment E, the synthesized mediator M issuch that when building the parallel composition $P||Q||E||M$, P and Q are able to communicate and coordinate by reaching their final states under the hypothesis of fairness.

2.2.2 Auxiliary Definitions

In the following we provide some auxiliary definitions needed to describe the theory. The first definition illustrates the conditions under which two actions $<a_1,In_1,Out_1>$ and $<a_2,In_2,Out_2>$, labeled with ontological concepts, are compatible. In particular, two complementary (input/output) actions are compatible if and only if (i) for each input expected by the provided action, its data type is subtype of/equal to ($\subseteq$) the

[4]We solve granularity mismatches on sequences of transitions without branches on the minimal eLTS.

type of some input data produced by the required action ($In_2 \subseteq In_1$); and (ii) for each output expected by the required action, its data type is subtype of/equal to the type of some output data produced by the provided action ($Out_1 \subseteq Out_2$). It is worth to notice that the following definition is intended to *actions, with data, with aligned (same) alphabet*.

Definition 5 (*Compatible Actions*) *Let* $<a_1,In_1,Out_1>$ *and* $<a_2,In_2,Out_2>$ *be two complementary actions where* $a_1 = act$ *and* $a_2 = \overline{act}$*. We say that* a_1 *and* a_2 *are compatible actions, denoted by* $a_1 =_C a_2$ *iff* $\forall i_j \in In_2\ \exists i_i \in In_1 \mid i_j \subseteq i_i \wedge \forall o_i \in Out_1\ \exists o_j \in Out_2 \mid o_i \subseteq o_j$*.*

We use the shorthand notation $a_1 =_C a_2 \Leftrightarrow In_2 \subseteq In_1 \wedge Out_1 \subseteq Out_2$ to denote the same.

The initial state together with the final states, define the boundaries of the protocol's coordination policies or traces. A **coordination policy** or **trace** is indeed defined as any sequence of actions that starts from the initial state and ends into a final state. It captures the most elementary behaviors which are meaningful from the user perspective. Then, a coordination policy or trace represents a communication (i.e., coordination or synchronization) unit and is formally defined as follows.

Definition 6 (*Trace* or *Coordination Policy*) *Let* $P = (S, L, D, F, s_0)$ *be an eLTS. A trace* $t = l_1\ l_2\ \ldots\ l_n \in L^*$ *is such that:* $\exists (s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2\ \ldots\ s_m \xrightarrow{l_n} s_n)$ *where* $\{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F$*. We use the usual compact notation* $s_0 \stackrel{t}{\Rightarrow} s_n$ *to denote a trace, where* t *is the concatenation of labels of the trace.*

We recall that, in our setting, protocols P and Q, in order to communicate, need to be composed in parallel with a mediator M that let them interact by performing some mediation between them. Thus each protocol will synchronize directly with the mediator that will perform the coordination between P and Q by also taking into account the environment. This implies that the actual interaction happens between an application protocol and a mediator, i.e., P, M and Q, M respectively.

2.2.3 Updated Theory

This section provides an updated formalization of our Abstract CONNECTor Synthesis including in particular: the *Abstraction - Behavioral Matching - Mediator Synthesis* process. We recall that we consider: two systems in the same domain of interest with their respective eLTSs; a widely shared application domain ontology conceptualizing actions and data of the domain where each application action and data refers/maps to some concept in the application domain ontology.

Abstraction: Identification of the common Language

The aim of this phase is to make models comparable and, if possible, to reduce their size thus allowing to ease and speed up the subsequent reasoning on them. The subsequent step will check whether the protocols we are considering, within the same application domain, are compatible. If the protocols are compatible, this imply that at a given level of abstraction, there exists a common language that can be identified. We recall that the input to this step are the application domain ontology and two NSs models each including: its own ontology that is a subset of the application domain ontology; its own eLTS that describing the ontology-based behavioral model. Hence *identifying the common language* means checking the existence of a kind of *intersection* among the ontologies of the two NSs that, when applied, makes them homogeneous. In particular, the check is about the existence of a concept o in the application domain ontology O of protocols P and Q, that in the same time is in relation $\mathcal{R}$ with two subsets of ontological concepts $\{o_i \ldots o_n\}$ of P and $\{o_1 \ldots o_m\}$ of Q. More formally:

Definition 7 (*Common Language*) *Let:*

- P *and* Q *be protocols,*
- $O = (L, A)$ *be the application domain ontology of both* P *and* Q*,*
- $O_P \subseteq O$ *and* $O_Q \subseteq O$ *be the ontologies of* P *and* Q *respectively;*

- $\mathcal{R}$ *is a relation among ontology concepts;*

The common language between P *and* Q *is a set* $O_{PQ} = \{o \in O \mid \exists\, S_P = \{o_i, \ldots, o_n\} \in O_P, |S_P| \geq 1 \wedge \exists S_Q = \{o_j, \ldots, o_m\} \in O_Q, |S_Q| \geq 1 : S_P\, \mathcal{R}\, o \wedge S_Q\, \mathcal{R}\, o\,\}$

Note that in the definition above $|S_P|$ ($|S_Q|$ resp.) can be equal to 1. In this case the set $\{o_i, \ldots, o_n\}$ ($\{o_j, \ldots, o_m\}$) is made up by one concept let us say o_i (o_j resp.) that *may coincide with* o.

Figure 2.6 shows the application domain ontology where the boxes with gray striped background highlight the application ontology of Blue of our case study while the boxes with gray background highlight the application ontology of Moon. In other words, the boxes show the result of the classification of the Blue's ontology and of the of Moon's ontology.

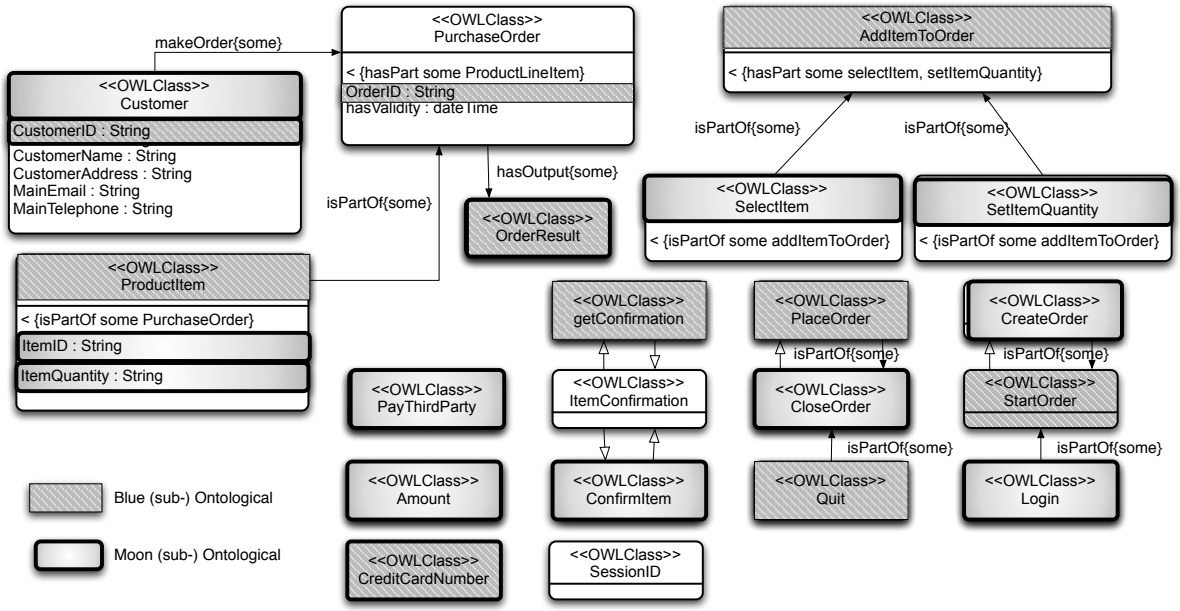

Figure 2.6: Classification of Blue and Moon application's ontologies

Figures 2.7 and 2.8 show the abstract protocol of Blue and of Moon respectively.

Since the common language does not take into account the input/output direction of actions, an action act belonging to the common language might be the outcome of several pairs of labels. For instance, both pairs $< act\ , In\ , Out >$, $< \overline{act}\ , In\ , Out >$ and $< act\ , In\ , Out >$,$< act\ , In\ , Out >$ would lead to the action act. Still, since the common language identification does not take into account the actions direction, the behavioral matching phase have to check that the protocols over the common language have complementary traces.

Together with the identified common language, systems may possibly also have non-common language. The non-common actions, e.g., interactions with third parties and extra actions have to be managed by the mediator while common-actions have to be exchanged among systems -through the mediator. Nevertheless systems have the same language, they can still suffer of *ordering mismatches* and/or incompatible traces due for instance to non-compatible actions direction input/output. For instance the two actions $<op,In,Out>$ and $<op,In,Out>$ are clearly incompatible.

Thus, the aim of the subsequent step is to check whether the two abstracted protocols have at least a *complementary coordination policy*, i.e., whether the abstracted protocols may in fact synchronize at least on a trace.

Behavioral Matching

The behavioral matching step aims at checking the NS applications compatibility by identifying compatible behavior (traces) modulo ordering mismatch, third parties interactions and extra actions. A successful

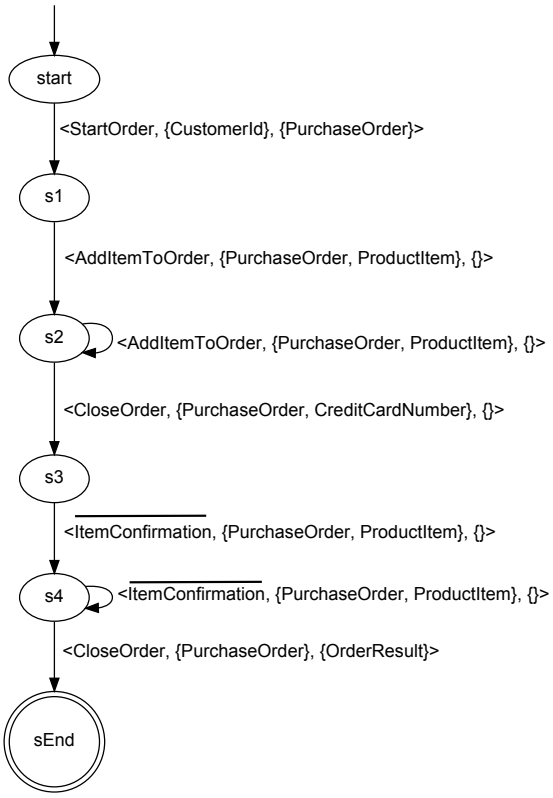

Figure 2.7: Abstract Blue protocol

matching implies that a mediator exists and is automatically synthesized by the subsequent phase. The output of the matching includes: (i) the degree and direction of compatibility; (ii) the compatible set of traces –identifying a sub-eLTS – labeled by actions and data concepts of the domain ontology including τs for third parties actions; (iii) a mediator as an eLTS defined over the domain-specific ontology including τ that realizes a skeleton that then needs to be refined to become a mediator. Figure 2.9 shows the auxiliary mediator that is one of the output of the matching.

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 abstract protocols A_P and A_Q based on their common language O_{PQ} we need to check that the abstracted protocols A_P and A_Q have complementary coordination policies. To do this, we use the *behavioral compatibility* relation between A_P and A_Q, which succeeds if A_P and A_Q have *a set of pairs of complementary coordination traces*, i.e., at least one pair.

Before introducing the behavioral compatibility relation, let us define the *complementary coordination policies* based on *complementary actions with data*. Informally, two coordination policies are complementary if and only if they are two sequences of complementary actions with data possibly in different order for which it exists a mediator that allows them to correctly synchronize by realizing the needed reordering. That is, traces t and t' are complementary if and only if: each action of t has its complementary action in t' and viceversa with switched roles among t' and t and it exists a mediator that allows the traces to successfully synchronize by reaching their final states. More formally:

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ *and* $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$*,*
- A_P, A_Q *be the abstracted protocols of* P *and* Q *respectively,*
- T_P *and* T_Q *be the set of all the traces of* A_P *and* A_Q*, respectively,*

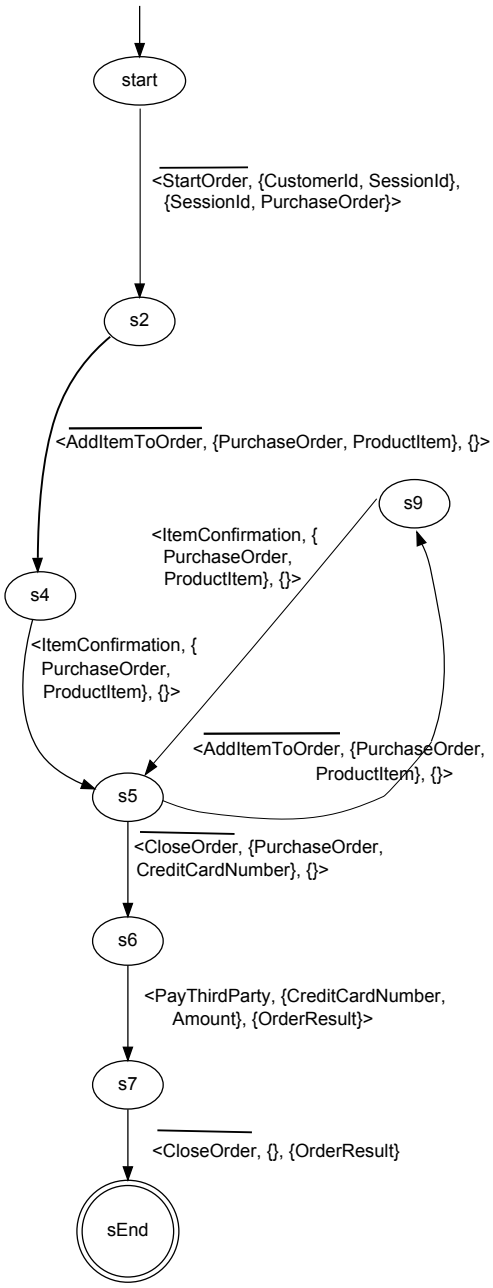

Figure 2.8: Abstract Moon protocol

- $t = l_1\ l_2\ \ldots\ l_n \in T_P$ *and* $t' = l'_1\ l'_2\ \ldots\ l'_m \in T_Q$.

Coordination policies t *and* t' *are complementary coordination policies iff the following conditions hold (discarding the* τ*s and extra actions):*

(i) it exists a bijection among $l_i \in t$ *and* $l'_j \in t'$ *:* l_i *and* l'_j *are complementary actions;*

(ii) it exists a mediator that allows t *and* t' *to synchronize successfully, i.e., managing reordering, extra actions and* τ*s allows the traces to reach their final states without deadlock while exchanging data effectively.*

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 because we work under the assumption of causal independence of actions. Hence, two traces having all complementary labels (skipping the τs) but in different order are considered to be complementary coordination policies (modulo a reordering). Therefore, while doing this check, we store such information that will be used during the mediator synthesis in addition to other information, e.g., the abstraction information.

As said above, we perform the complementary coordination policies check on the abstracted protocols A_P and A_Q, which are expressed in the common language plus τs representing third parties synchronization. We further use the *behavioral matching relation* or *compatibility* to describe the conditions that have to hold in order for two protocols to be compatible. Formally:

Definition 9 (*Behavioral Matching or Compatibility*) *Let:*

- P *and* Q *be protocols,*
- A_P *and* A_Q *be the abstract protocols(i.e., on the same alphabet) of* P *and* Q *respectively,*
- t_i *be a coordination policy of* A_P *and* t'_i *be a coordination policy of* A_Q.

Protocols P *and* Q *have a behavioral matching or are compatible iff there exists a set* C *of pairs* (t_i, t'_i) *of complementary coordination policies of* A_P *and* A_Q *where* $|C| \geq 1$.

The behavioral matching relation defines necessary conditions that must hold in order for a set of NSs to interoperate through a mediator. In our case, till now, the set is made by two NSs and the matching condition is that they have at least a 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, as mentioned previously, protocols can have a set of pairs of complementary traces. We then define four different levels of behavioral matching or compatibility, spanning from non-matching, to partial- and total-matching:

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

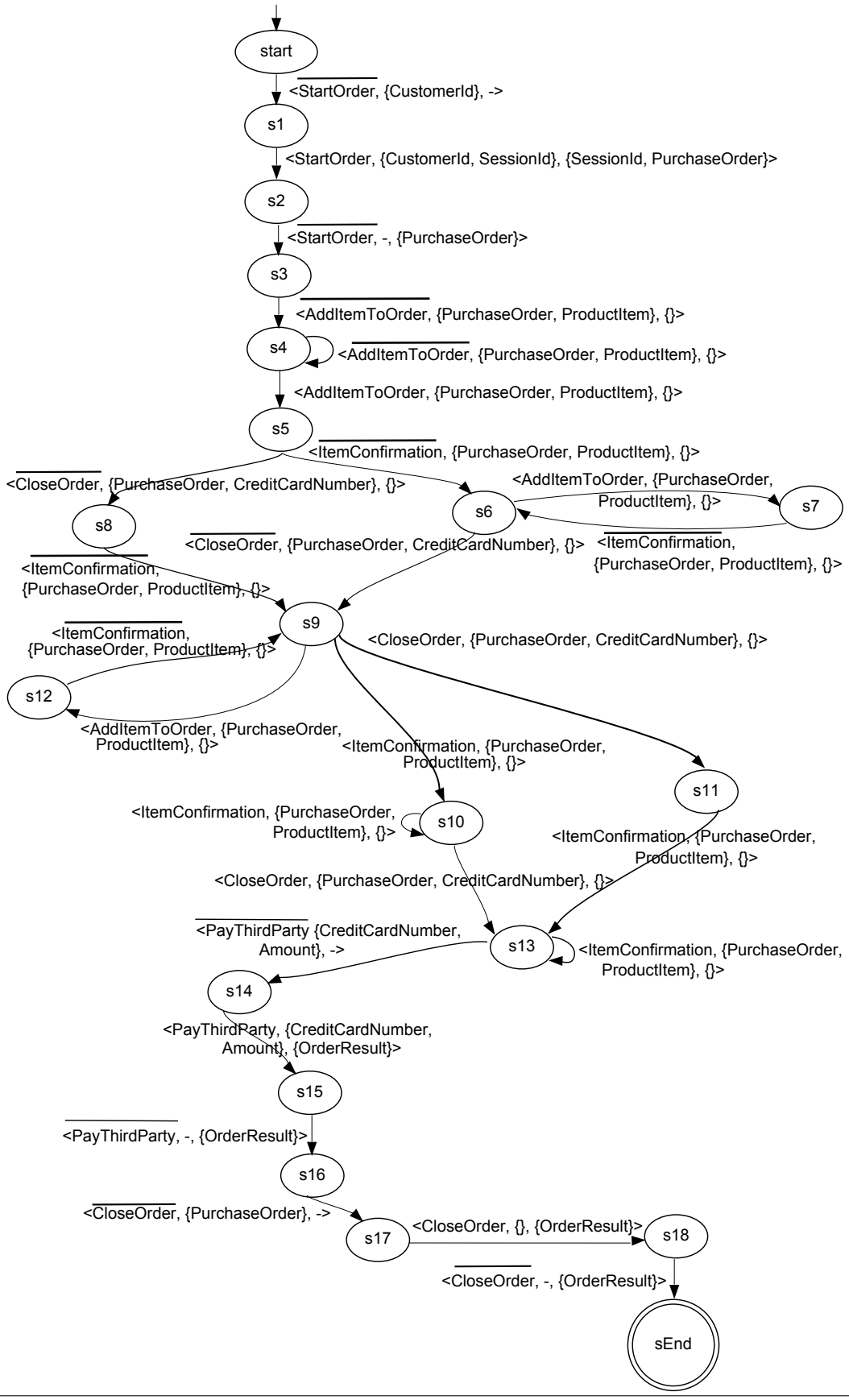

Figure 2.9: Auxiliary mediator protocol

Mediator Synthesis

The Mediator Synthesis produces a mediator that addresses the mismatches found during the previous phase so as to allow the NSs to communicate and coordinate. We recall that during the behavioral matching phase *mediators traces* are found that all together represent the skeleton of the mediator to be synthesized. During the mapping phase this skeleton of mediator needs to be refined thus producing a mediator over the original protocols languages. The input of the synthesis is then the output of the behavioral matching step and the output is a correct-by-construction mediator.

Given two compatible protocols P and Q and the set C made by their pairs of complementary coordination policies, we want to synthesize a mediator M such that the parallel composition $P||M||Q||E$ where E is the environment, allows P and Q (i.e., their portion C) 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 language. 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. We recall that the synchronization cannot be direct since the mediator needs to perform suitable manipulations as for instance actions reordering or translation according to the ontology mappings used to align protocols and to identify the common language.

As we said previously, we work on traces instead of working on protocols, hence producing a set of mediating traces for C where we recall that the traces of C's pairs are traces on the abstract protocols A_P and A_Q. Then, the mediator protocol AM for C can be easily obtained by merging the mediating traces. AM can be considered an "auxiliary mediator" since it mediates between abstract protocols. To obtain the corresponding "mediator", we then need to translate each abstract action to its corresponding concrete (sequence of) action(s), i.e., on the languages of P and Q.

Therefore, an auxiliary mediator is a protocol that, for each pair of complementary coordination policies $c_{pq} = (c_p, c_q)$ in C, builds a mediating trace m_{pq} such that, for each action $<op,In,Out>$ or also τ in c_p and in c_q it always first receive the action and then properly resend it. We recall that an action $<op,In,Out>$ can be equivalently expressed by the following two primitives $<op,In,->$ and $<op,-,Out>$. This applies similarly for $<\overline{op},In,Out>$. An abstract mediator is thus more formally defined as follows.

Definition 10 (*Mediator*) *Let:*

- A_P *and* A_Q *be the abstract protocols of protocols* P *and* Q *respectively;*
- $c_p = l_1 \dots l_n$ *and* $c_q = l_1 \dots l_m$ *be traces over* A_P *and* A_Q *respectively;*
- C *be the set of pairs of complementary traces between* A_P *and* A_Q *and* $(c_p, c_q) \in C$ *;*

The mediator M *for* C *is defined as follows:*
$M = \{m_{pq} = l_1 \dots l_k,\ k \geq n+m \mid$ *trace* m_{pq} *is such that* $\exists\ \{l_i = <\overline{op},In,->,\ l'_i = <op,In,Out>,\ l''_i = <\overline{op},-,Out>\},\ 1 \leq i < i' < i''\ \forall$ *pair of compatible actions* $l_h, l'_h : l_h = <op,In,Out> \in c_p$ *(*$\in c_q$ *resp.)* $\wedge\ l'_h = <\overline{op},In,Out> \in c_q$ *(*$\in c_p$ *resp.)* $\}$

We recall that the auxiliary mediator described above is intended to mediate among the abstract protocols A_P and A_Q. Thus, we need to translate each abstract action to its corresponding concrete (sequence of) action(s), i.e., on the languages of P and Q to obtain the corresponding mediator.

Figure 2.10 illustrate the mediator for our case study. The mediator is logically made up of two separate components: M_C and M_T. M_C speaks only the common language and M_T speaks only the third parties language. M_C is an eLTS 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.

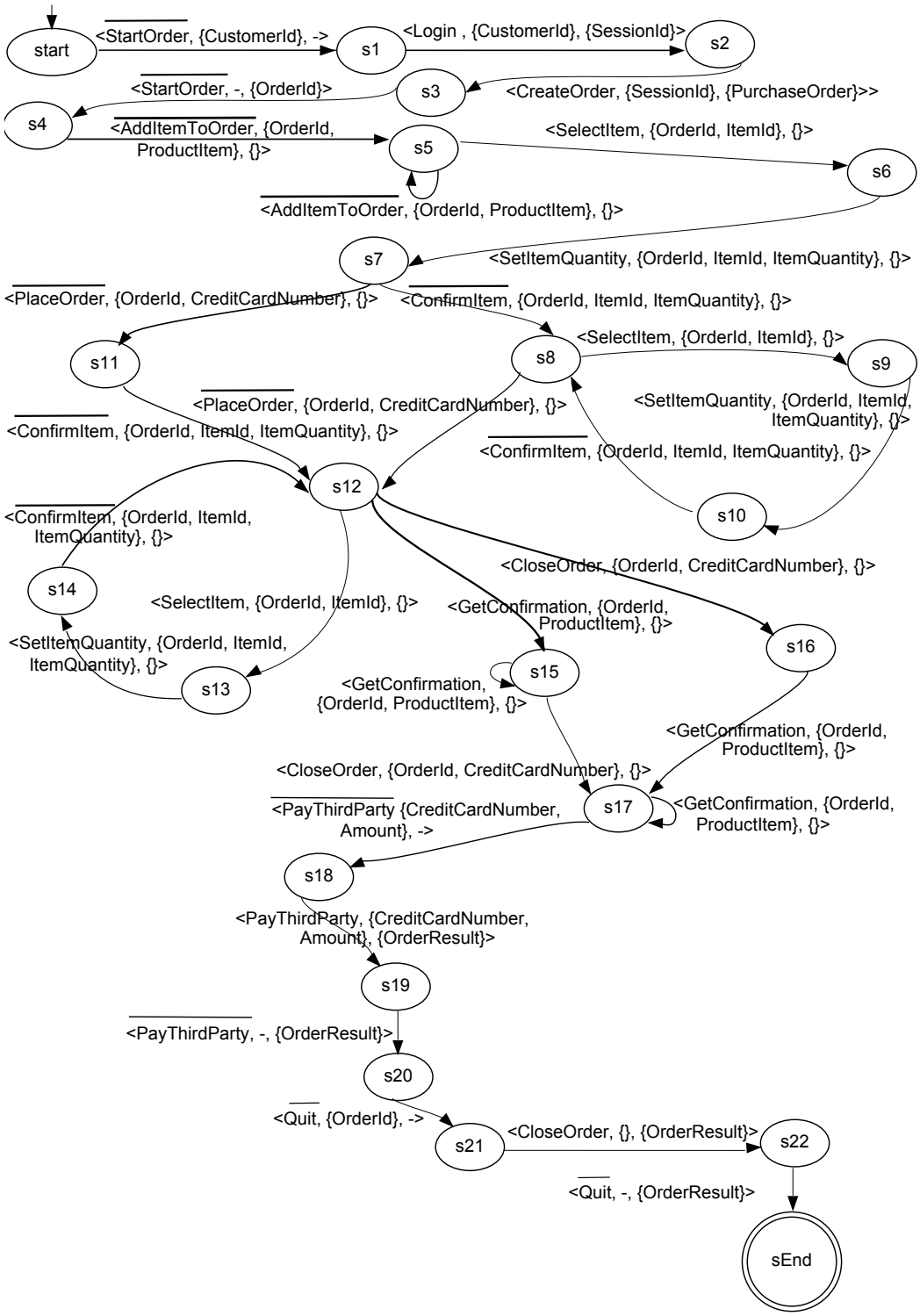

Figure 2.10: Mediator protocol

2.3 Correctness

In order to informally state correctness of the WP3 synthesis process, we can directly exploit, and hence refer to, the work done during Year 3 by Work Package WP2.

More specifically, as described in details in Deliverable D2.3 [6], the work carried on within WP2 during Year 3 led to the definition of a component and connector specification theory. This theory allows us to characterize the WP3 *mediator synthesis problem* as a suitable quotienting problem, in which ontological constraints and an abstract goal specification are taken into account. In particular, in Deliverable D2.3, relationships between the specification theory of WP2 and the mediator theory of WP3 are shown, thus devising a suitable mapping from the mediator theory of WP3 to the WP2 specification theory. This allows the work of WP3, described in this chapter, to inherit the interesting results of the WP2 specification theory. More verbosely, as discussed in [6], it allows one to state that a mediator synthesized *à la* WP3 is *correct*, i.e., it does not introduce inconsistencies, e.g., communication mismatches, deadlocks, etc., and is *most general*, i.e., any other mediator is a refinement of it.

To show that the WP3 mediator theory fits in with the WP2 specification theory, the authors of Deliverable D2.3 give a semantic preserving mapping from the former to the latter. As a direct consequence of that, the two fundamental operations of mediator synthesis, *protocol matching* and *protocol mapping*, can be abstractly characterized as a suitable quotienting problem and related synthesis algorithm, by exploiting the notions of *quotient*, *ontological constraints* and *goal* as shown in Deliverable D2.3.

2.4 Conclusion

Interoperability problems between heterogeneous protocols are the focus of our work. In this chapter we concentrated on the Abstract CONNECTor Synthesis, including three phases: Abstraction (Identification of the Common Language), Behavioral Matching, and Mediator Synthesis. We provided a revised CONNECTor Theory to synthesize a CONNECTor between compatible protocols that is supported by two prototypal implementations illustrated in Chapters 3 and 4.

The theory, with respect to the one presented last year, (i) explicitly models the data that the observable application actions convey and (ii) provides a general description of the manipulations needed all along the abstraction and behavioral matching phases that are the costly and hard ones while in the previous version they were more implementation specific.

The CONNECTor produced by our theory handles mismatches and third parties conversations considering the traces of each networked system as goals that have to be met through interactions with the other NS. Hence, by performing the behavioral matching check, our approach let the goal of the communication emerge. When working on infinite protocols, i.e. protocols with cycles, we handle a finite set of the the infinite set of traces that can be generated by considering that cycles are executed at most a fixed number of times.

For the future, we plan to investigate and extend the theory with the following aspects: to take as input a goal specification together with the current input; to design refined behavioral matching and synthesis algorithms; to try to build the common language on the fly instead of classifying ontologies a priori.

In the subsequent chapters, two different implementations of the Abstract CONNECTor Synthesis theory are presented: Goal-based abstract CONNECTor synthesis (Chapter 3) and Mapping-driven Abstract CONNECTor Synthesis (Chapter 4). The two approaches are then compared in Chapter 5, where their differences and peculiarities are highlighted, while Chapter 6 provides a unified interface for them.

3 Goal-based abstract CONNECTOr synthesis

One of the main focuses of CONNECT is, as stated in Chapter 2, to develop the theory and the practice needed to synthesize automatically a CONNECTOr (i.e. a mediator) capable of enabling the communication between two networked systems. In last year deliverable D3.2 [9] and in Chapter 2 of this deliverable, a theory of mediators has been developed and refined.

In this chapter we propose a practical implementation for the abstract CONNECTOr theory, presented in Chapter 2. Our implementation tackles the problem of creating an abstract CONNECTOr automatically, given a *pair* of networked systems. The CONNECTOr we produce is defined abstract since it needs to be concretized, as described in Deliverable D1.3 [4], in order to be executed.

Literature has been tackling this problem in several ways in recent years, proposing a range of solutions that go from those able to find mismatches between two components, without solving them (as for instance [35] and [34]), to those capable of suggesting possible mismatches solution, but needing human assistance in order to create a mediator (i.e. [40]), to more complete and automatic solutions (e.g. [20], [56]). However the aforementioned solutions do not consider in the mediator synthesis process that two networked systems are usually connected with a given goal in mind. This aspect is important in CONNECT, as user goals are used to discover and select the networked systems that should communicate (see D1.3 [4] for details about goals and how they are used in discovery and selection).

The problem of implementing a framework for automatic mediator synthesis has been tackled also in D3.2 [9], producing a solution that is able to handle simple mismatches (i.e. signature mismatches, described in the mismatch models proposed in [28] and [71]). Also this preliminary solution, just like the previously mentioned ones, does not consider the goal driving the connection of two networked systems.

In this chapter we provide a possible implementation of the CONNECTOr theory, specifying an automatic solution to the problem of synthesizing an *abstract* CONNECTOr to enable the communication of two networked systems, triggered by a user goal. The produced CONNECTOr is defined abstract as it disregards the middleware on which the two networked systems are actually implemented and considers only their application protocols (see Chapter 2 and D3.2 for definitions and details).

The solution presented in this chapter is based on what proposed in [26, 15]. Here we extend what presented in those works, introducing a well defined ontology matching process (see Section 3.3), adding the support for goal-based CONNECTOr synthesis (see Section 3.4), and extending the communication model that allows us to synthesize mediators to take into account communication models different from the client-server one, the only one considered in [26, 15] (see Section 3.4).

The rest of the chapter is organized as follows: Section 3.1 presents an overview of our approach, detailing the inputs and the outputs and relating our implementation to the theory in Chapter 2, Section 3.2 presents the simplificative hypotheses we consider to implement the goal-based synthesis of abstract CONNECTOrs, Section 3.3 describes how we implement the common language inference, introduced in Definition 7, Section 3.4 describes how we perform the abstract CONNECTOr synthesis phase, implementing the *Matching* introduced in Section 2.2.3, while Section 3.5, we show how our technique works on the case study described in Section 1.5. The tool we produced to evaluate the approach is explained in Section 3.6, through a description of the framework in which we implemented the approach, finally Section 3.7 draws some conclusions and proposes some possible future research directions.

3.1 Synthesis Process Overview

The process for synthesizing an abstract CONNECTOr, presented in this chapter, is an implementation of what presented in Chapter 2. The current implementation restricts the theory considering only the synthesis of connectors for a couple of networked systems (i.e. no third parties interactions are considered). Our synthesis process takes the following inputs:

- A $Connection = (Req, Concept, G, Prov)$ between a required *networked system description* Req and a provided networked system description $Prov$, proposed by the discovery enabler (see Chapter 3 of Deliverable D.1.3 for details about discovery enabler and connections). A network system description is defined as a tuple $L \times 2^{Affordance}$, where L is defined as in Definition 3 $Affordance = OntologyConcept \times eLTS$. For the sake of simplicity, in the rest of the chapter we will refer to $eLTS$

as *affordance protocol* or *simply protocol*. Moreover in a $Connection$, G is the user goals that the given connection should satisfy and $Concept$ is the affordance of Req and $Prov$ the connection refers to.

- A domain specific ontology used to annotate each of the two affordance models. The annotations associate each action in the set of actions with data of each protocol to an ontology concept. Moreover the annotations also associate each input and output parameter of an action to an ontology concept. As previously defined in Deliverable D1.2 [3], we assume that annotations are provided using SAWSDL [78].

Our abstract CONNECTor synthesis process goes through two main phases:

1. *Common Language Inference*: this phase partially implements the abstraction phase described in Section 2.2.3, inferring a common language for the two input protocols, through an *alignment* process on their alphabets (see Section 3.3).

2. *Matching*: this phase implements what theorized in Section 2.2.3, synthesizing an abstract CONNECTor M that enables the communication between the networked systems, represented by the affordances in the $Connection$ given as input to the approach. The abstract CONNECTor should ensure that the communication between the two systems can take place (i.e. a *feasible interaction* between the two systems is possible, as defined in Section 3.4). Moreover the CONNECTor should ensure that the communication satisfies user goals in the $Connection$. This is achieved by ensuring that $P_{req} \times M \times P_{prov} \models G$. The synthesis phase is performed through SMT-based model checking on the two ontology-aligned protocols, and is explained in Section 3.4.

In the rest of this chapter we are going to detail the two phases of our synthesis process, supporting the explanation by the application of our technique to the case study in Chapter 1.5.

3.2 Simplificative Hypotheses and Case Study Refinement

Here we refine the case study introduced in Section 1.5. The refinement is necessary because, as previously mentioned in this chapter, our implementation makes some restrictive hypotheses with respect to the theory, introduced in Chapter 2.

The main hypothesis introduced by the goal-based abstract CONNECTor synthesis is that the process will work on two networked systems only. This hypothesis was introduced to simplify the implementation of the synthesis enabler. Consider for instance the networked system protocol reported in Figure 2.4. The part in which the service handles the client payment presents a third party interaction. In the *Moon* customer service representation in that Figure, the service receives the data from the client, but then it needs to interact with a third party service (i.e. a payment hub) in order to validate the received data and to proceed with the withdrawal of the needed amount of money from the customer credit card. This happens through the invocation of the $< PayThirdParty, \{CreditCardNumber, Amout\}, \{OrderResult\} >$ action. In our approach third parties interactions are disregarded, consequently this interaction will be simplified, as shown in Figure 3.1.

There is also a second hypothesis that is going to be taken into consideration in this chapter: a networked system can only read data produced by the other system, and not by itself. The introduction of this hypothesis is motivated by our reasoning mechanism to check the functional matching of networked systems protocols traces (see Section 3.4), and will be explained in more details later in the chapter. Consider for instance the $< \overline{Login}, CustomerId, SessionId >$ action in the *Moon* customer service, reported in Figure2.4, and its successor $< \overline{CreateOrder}, \{CustomerId\}, \{SessionId\} >$. The $\overline{CreateOrder}$ action expects to receive as input the SessionId parameter, which is not sent by any action in the *Blue* client in the example (see Figure 2.3). The parameter should come from the output of the $\overline{Login}$ action, however, with our second simplificative hypothesis, the parameter cannot be read by those outputted by the same networked system, so generating a CONNECTor in this case would not be possible. For this reason, we disregard the SessionId parameter, as shown in Figure 3.1. In the rest of the chapter the protocol shown in the latter figure will be used as reference for the *Moon* customer service.

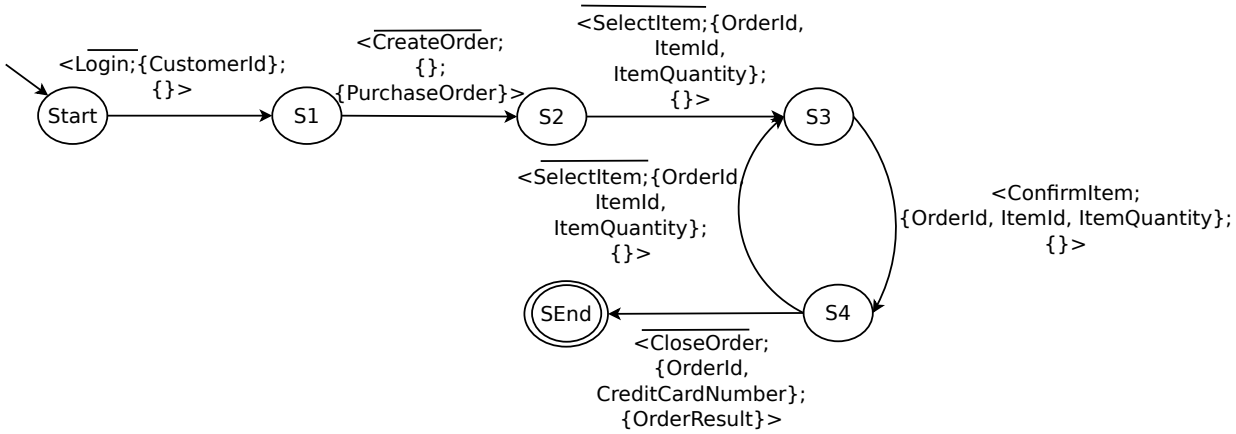

Figure 3.1: Refined version of the *Moon* customer service networked system.

Finally we introduce a third simplification about the semantics of the received parameters: each received parameter can be *read just once* by the networked system that receives it, and then is discarded by the mediator. We introduce this conservative hypothesis, as the language introduced in last yeard deliverable D3.2 [9], does not distinguish at the moment the cases in which a parameter should be read only once or can be read more times. This hypothesis calls for a refinement to be introduced in the *Moon* service protocol. Consider for instance the sequence of actions $< \overline{SelectItem}, \{OrderId, ItemId\}, \{\} >$ and $< \overline{SetItemQuantity}, \{OrderId, ItemId, ItemQuantity\}, \{\} >$ in the *Moon* service in Figure 2.4. Each of the actions in that sequence requires to read an instance of each of its input parameter. When the parameter is read, following our hypothesis, The action that can provide instances of the same parameters in the *Blue* client is the $< AddItemToOrder, \{OrderId, ProductItem\}, \{\} >$ action. Since, as shown in the ontology, the $ProductItem$ input parameter of this action is composed of $ItemId$ and $ItemQuantity$, it provides enough input parameters to invoke the $\overline{SelectItem}$ operation, but, in our hypothesis, not enough to invoke also the following $\overline{SetItemQuantity}$ operation on the *Moon* service. In order to avoid this problem, we refine the $\overline{SelectItem}$ in the *Moon* service as shown in Figure 3.1.

3.3 Common Language Identification

Common language identification is the process that *aligns* the alphabets L_{req} and L_{prov}, to a common alphabet, as specified in the theory, in Section 2.2.3. We here implement the common language identigication by taking as input the two alphabets L_{req} and L_{prov}, and a domain specific ontology,identifying the common alphabet L_M, and substituting L_{req} and L_{prov}, with the identified common alphabet (i.e. what we call alignment process). In practical terms the alignment process is necessary because, in order to perform model checking on P_{req} and P_{prov}, we need to have them defined on a common alphabet.

In order to be able to perform the matching on the domain specific ontology, we need that the latter contains the two following relation types:

- *subclass* relations,
- *part-whole* relations existing between an element and its parts, represented by the *hasPart* and its inverse *partOf* relations.

Existing OWL reasoners, usually, try to infer subclass relations from other existing ontology relations [68]. Anyway currently available reasoners do not take into account part-whole relations [62], as stated in W3C best practice recommendation [79]. To effectively work-around this limitation we refer to the latter recommendation, and we assume that the ontology comes with the *hasPart* and the *partOf* relations.

The process of inferring a common language for the two input networked systems is composed of two steps.

- *Action alignment*: given an action A_1 in L_{req} this step tries to find a concept OC, in the domain specific ontology used as input for the synthesis process, that can be used to align A_1 to one or more actions in L_{prov}.The alignment process considers the part-whole and the subclass relations in the ontology in the following order.

  1. Part-whole relations, A_1 is involved in are considered in a first moment. In case A_1 is in at least one *hasPart* relation with ontology concepts $A_2, \ldots, A_k$, in L_{prov}, these actions are considered for alignment. The alignment replaces the annotation $A_2, \ldots, A_k$ with the annotation A_1 .

     Consider for instance the *StartOrder* action in the *Blue* client (as represented in Figure 2.3). The concept annotating this action, as reported in the ontology in Figure 2.2, is in a *hasPart* relation with the *Login* and *CreateOrder* concepts. These two concepts annotate two actions in the *Moon* customer service. In this case we can align the actions annotated by the *Login* and *CreateOrder* concepts, on the *StartOrder* concept. Consequently the first two annotations are substituted by *StartOrder*.
  2. Subclass relations are considered. In case no part-whole relation exists for a concept A_1 annotating an action in L_{req} we search for an ontology concept OC such that $A_1 \sqsubseteq OC$ (to be read as A_1 is subsumed by OC, i.e. A_1 is subclass of OC, according to the definition given in [58]), and $A_2 \sqsubseteq OC$. It is worth nothing that several concepts $OC_1, \ldots, OC_k$ can exist, such that $A_1 \sqsubseteq OC_1, \ldots, A_1 \sqsubseteq OC_k$ and $A_2 \sqsubseteq OC_1, \ldots, A_2 \sqsubseteq OC_k$. In this case we consider the hierarchy defined in the domain specific ontology for the classes $A_1, A_2, OC_1, \ldots, OC_k$ and we chose the class OC, having the lowest distance in the hierarchy from A_1 and A_2.

     If a concept OC exists, the annotations A_1 and A_2 are replaced by that concept. Consequently, the two actions, previously annotated by concepts subsumed by OC, are now annotated by the same concept, and are defined as *aligned*.

     Consider for instance the action annotated by the $GetConfirmation$ concept in the *Blue* client alphabet. As reported in the ontology in Figure 2.2, the $GetConfirmation$ is a subclass of the $ItemConfirmation$ concept. The latter concept has also another subclass in the domain specific ontology: the $ConfirmItem$ concept. $ConfirmItem$ annotates an action in the *Moon* customer service alphabet. Consequently the actions respectively annotated by $GetConfirmation$ and $ItemConfirmation$ can be aligned on the $ConfirmItem$ concept.

  At the end of this first step, the annotations of the actions in L_{Req} and in L_{Prov} are aligned.

- *Parameter alignment*: given two actions annotated respectively by A_1 and A_2, aligned on an ontology concept OC, this step considers their parameters. The alignment of parameters of two aligned actions works in a way similar to the action alignment, with the only difference that we assume that concepts representing parameters in the ontology can also have data properties, as shown in Figure 2.2. The presence of data properties, implies that some data parameters of the actions in L_{req} and L_{prov}, can be annotated with those data properties, instead of the concepts. For instance, in Figure 3.1, the unique input parameter of the output action $\overline{Login}$ is annotated with $CustomerId$, which is a data property of the $Customer$ concept, as shown in Figure 2.2. For what concerns our approach, the latter case is treated as if a part-whole relation existed between the concept and its data properties (e.g. in the aforementioned example there exists a part-whole relations between $Customer$ and $CustomerID$).

  The parameter alignment process, similarly to the actions alignment process, is divided in two steps.

  1. Considering a parameter of A_1, annotated with the concept OC_1, this step analyzes in a first moment its data properties and the part-whole relations, OC_1 is involved in. In case OC_1 is in at least one *hasPart* relation with ontology concepts $OC_2, \ldots, OC_k$ (or respectively has $OC_2, \ldots, OC_k$ as data properties), and if $OC_2, \ldots, OC_k$ annotate some of the parameters of A_2, the second step of the process considers them for the alignment. The alignment replaces the annotation OC_1 by the annotations $OC_2, \ldots, OC_k$. In this way the OC_1 is substituted by those concepts that the ontology points out being its parts.

     In order to complete the alignment, it is necessary to consider that the parameter's semantics is also determined by the action it belongs to. To allow the semantics of a parameter to be

related to the actions it belongs to, during the alignment, a reference to the action concept is added to the parameter annotation. Consequently, in the aforementioned case, the annotation OC_1 is replaced by $OC.OC_2, \ldots, OC.OC_k$, where OC is the concept aligning A_1 and A_2. In the same way, the $OC_2, \ldots, OC_k$ parameters of A_2 are substituted by the annotations $OC.OC_2, \ldots, OC.OC_k$. At the end of this step the parameters OC_1 and $OC_2, \ldots, OC_k$, are *aligned* on the annotations $OC.OC_2, \ldots, OC.OC_k$.

Consider again the case study introduced in Chapter 1.5, and the actions $GetConfirmation$ in the *Blue* client alphabet and $ConfirmItem$ in the *Moon* customer service alphabet. These two actions were aligned on the $ItemConfirmation$ concept. Let us consider their parameters. $GetConfirmation$, for instance has a parameter annotated by the $ProductItem$. As shown in the ontology in Figure 2.2 the $ProductItem$ has the following data properties, $ItemId$and $ItemQuantity$. Those two data properties, annotate two output parameters of the action $ConfirmItem$. Following our procedure, we can align the annotation $ProductItem$ and the $ItemId$ and $ItemQuantity$ annotations, of the action annotated by $ConfirmItem$, with $ItemConfirmation.ItemId$ and $ItemConfirmation.Quantity$ annotations.

2. In case the parameter annotated by OC_1 is not in a part-whole relation, we consider OC_1 super-classes in the ontology. Given OC_1, and a parameter of A_2,annotated with the concept OC_2, this step tries to find a concept OC_3 such that $OC_1 \sqsubseteq OC_3$ and $OC_2 \sqsubseteq OC_3$. In case OC_3 exists, similarly to the previous case, the annotation OC_1 is substituted by the annotation $OC.OC_3$, and the annotation OC_2 is substituted by the annotation $OC.OC_3$, where OC is the concept aligning A_1 and A_2. At the end of this step the parameters OC_1 and OC_2, are *aligned* on the annotation $OC.OC_3$.

   Differently than from what considered for actions, when aligning parameters to an ontology superclass, there is a degenerative case that needs consideration, and in presence of which, our approach does not produce an alignment . This case is the one in which an input parameter OC_1 of an action $\overline{A_1}$ (i.e. $\overline{A_1}$ expects to receive OC_1), needs to be aligned on an input parameter OC_2 of a provided action A_2 (i.e. A_2 sends OC_2), and $OC_1 \sqsubseteq OC_2$, or that in which an output parameter OC_1 of an input action A_1 (i.e. A_1 expects to receive OC_1) needs to be aligned to the output parameter OC_2 of a required action $\overline{A_2}$ (i.e. $\overline{A_2}$ sends OC_2), and $OC_1 \sqsubseteq OC_2$. To understand why the alignment needs not to be produced in this case, consider that the parameters identified by the ontology concepts OC_2 in the previous statement, are sent by the action they are parameters for, while those annotated by OC_1 are expected to be received. Now, since $OC_1 \sqsubseteq OC_2$, according to the definition in [58], OC_1 type "requires the same or more" than OC_2 type. Consequently, when OC_2 has to be received in place of OC_1, not all the pieces of information in OC_1 may be provided.

   It is interesting to notice why this degenerative case does not occur for actions alignment. Actions in our approach are considered to give a semantics to parameters, since they do not exchange any piece of data by themselves. The exchange of data between two networked systems, instead, takes place through the exchange of parameters. Consequently this case needs to be considered when aligning parameters, while it does not occurr for actions.

Let us apply our mapping process to the case study reported in Chapter 1.5. The alignment for the actions in the alphabets of the*Blue* client and the *Moon* customer service, are reported respectively in Table 3.1 and 3.2.

At the end of the alignment process the two alphabets, $L_{P_{prov}}$ and $L_{P_{req}}$, are *aligned* on a common alphabet L_M. In Section 3.3 we will describe how we can synthesize a CONNECTOr for P_{req} and P_{prov} whose alphabets are aligned on L_M. For the sake of simplicity, we are going to use the common language reported in Table 3.1 and in Table 3.2, instead of the original annotations of the actions, to support the explanation.

3.4 Abstract CONNECTOr Synthesis

Let us define our problem of finding a CONNECTOr M, such that, given a $Connection$ composed of a required protocol P_{req}, a provided protocol P_{prov} and some user goals $(G_1, \ldots, G_u)$, we have that $P_{req} \times$

<Action,{Input parameters},{Output parameters}>	Alignment
$< StartOrder, \{CustomerID\}, \{OrderId\} >$	$StartOrder, \{StartOrder.CustomerID\}, \{StartOrder.OrderId\} >$
$< AddItemToOrder, \{OrderId, ProductItem\}, \{\} >$	$< AddItemToOrder, \{AddItemToOrder.OrderId, AddItemToOrder.ItemId, AddItemToOrder.ItemQuantity\}, \{\} >$
$< PlaceOrder, \{OrderId, CreditCardNumber\}, \{\} >$	$< CloseOrder, \{CloseOrder.OrderId, CloseOrder.CreditCardNumber\}, \{\} >$
$< \overline{GetConfirmation}, \{OrderId, ProductItem\}, \{\} >$	$\overline{ItemConfirmation}, \{ItemConfirmation.OrderId, ItemConfirmation.ItemId, ItemConfirmation.ItemQuantity\}, \{\} >$
$< Quit, \{OrderId\}, \{OrderResult\} >$	$< CloseOrder, \{CloseOrder.OrderId\}, \{CloseOrder.OrderResult\} >$

Table 3.1: Common language inferred for the actions in the *Blue* client alphabet, reported in Figure 2.3.

< Action,{Input parameters},{Output parameters}>	Alignment
$< \overline{Login}, \{CustomerID\}, \{\} >$	$< \overline{StartOrder}, \{StartOrder.CustomerID\}, \{\} >$
$< \overline{CreateOrder}, \{\}, \{PurchaseOrder\} >$	$< \overline{StartOrder}, \{\}, \{StartOrder.OrderId, StartOrder.HasValidity\} >$
$< \overline{SelectItem}, \{OrderId, ItemId\}, \{\} >$	$< \overline{AddItemToOrder}, \{AddItemToOrder.OrderId, AddItemToOrder.ItemId, AddItemToOrder.ItemQuantity\}, \{\} >$
$< ConfirmItem, \{OrderId, ItemId, ItemQuantity\}, \{\} >$	$< ItemConfirmation, \{ItemConfirmation.OrderId, ItemConfirmation.ItemId, ItemConfirmation.ItemQuantity\}, \{\} >$
$< \overline{CloseOrder}, \{OrderId, CreditCardNumber\}, \{CloseOrder.OrderResult\} >$	$< \overline{CloseOrder}, \{CloseOrder.OrderId, CloseOrder.CreditCardNumber\}, \{CloseOrder.OrderResult\} >$

Table 3.2: Common language inferred for the actions in the *Moon* customer service alphabet, reported in Figure 3.1.

$M \times P_{prov} \models \bigwedge_{x \in [1,u]} G_x$.

A CONNECTor, is a piece of software that only allows *feasible interactions* between sequences of required actions $seq_{P_{req}} = (A_1, \ldots, A_i)$ and sequences of provided actions $seq_{P_{prov}} = (\overline{B_1}, \ldots, \overline{B_q})$, where $seq_{P_{req}}$ is a sequence of i actions allowed in the P_{req} protocol, while $seq_{P_{prov}}$ is a sequence of q actions allowed in the P_{prov} protocol.

The interaction between the two aforementioned sequences is feasible if:

- The input parameters expected by the actions in $seq_{P_{prov}}$ are a subset of the parameters provided by the actions in $seq_{P_{req}}$
- The output parameters provided by the actions in $seq_{P_{prov}}$ are a superset of the output parameters expected by actions in $seq_{P_{req}}$.

The rationale of this informal definition can be understood thinking of how the communication between two networked systems takes place. One of the two systems, the invoker, represented by $seq_{P_{req}}$ in our case, performs a sequence of provided actions. Each action, as stated in Definition 5, has a set of input parameters, which are provided by the invoker and sent over the network, and a set of output parameters, which the invoker expects to receive as a response to the performed action.

The other system, the invoked, in order for the interaction to be completed successfully, should perform a sequence of required actions, represented by $seq_{P_{prov}}$ in our case. Each of those required actions has a set of input parameters, that should be received from the network before the action is performed. In our model, consequently, a given action will produce the parameters contained in its output parameters set, when all the input parameters of a given action have been received.

In this perspective, requiring that the input parameters expected by the actions in $seq_{P_{prov}}$ are a subset of input parameters provided by the actions in $seq_{P_{req}}$ means guaranteeing that the input parameters of each action in $seq_{P_{prov}}$ will be provided by an action in $seq_{P_{req}}$. Moreover, requiring that the output parameters provided by the actions in $seq_{P_{prov}}$ are a superset of the output parameters expected by actions in $seq_{P_{req}}$ guarantees that the output parameters expected by the actions in $seq_{P_{req}}$ will be provided by the actions in $seq_{P_{prov}}$. An interaction respecting the aforementioned conditions is called *feasible*, and

represents the case in which there exists a functional matching between P_{req} and P_{prov},as stated in Definition 9 .

Feasible Interactions

Formally we can define the concept of *feasible interaction*, in the hypothesis that $seq_{P_{req}}$ is composed only of required actions and $seq_{C_{prov}}$ is composed only of provided actions, as follows[1]. For the trace $seq_{P_{req}}$ we consider the sets $\mathcal{D}_{P_{req}} = \bigcup_{1 \leq k \leq i} D_{A_k}$ and $\mathcal{E}_{P_{req}} = \bigcup_{1 \leq k \leq i} E_{A_k}$ respectively representing the set that contains all the annotations used to align A_k input parameter and the set containing all the annotations used to align A_k output parameters, for each A_k in $seq_{P_{req}}$. We similarly consider $\mathcal{D}_{P_{prov}} = \bigcup_{1 \leq h \leq q} D_{B_h}$ and $\mathcal{E}_{P_{prov}} = \bigcup_{1 \leq h \leq q} E_{B_h}$, which represent the set containing all the annotations used to align B_h input parameters and the set containing all the annotations used to align B_h output parameters, for each B_h in $seq_{P_{prov}}$.

Finally we define the two sets $\mathcal{D} = D_{P_{req}} \cup D_{C_{prov}}$ and $\mathcal{E} = E_{P_{req}} \cup E_{P_{prov}}$, containing all the annotations used to align input, respectively output, parameters that can be used in the two sequence, and consequently representing respectively all the input and output parameters that can be exchanged during an interaction.

For each action A_k in the required protocol, we define two sets of functions: $In_{A_k} : \mathcal{D} \rightarrow \{1, 0\}$, and $Out_{A_k} : \mathcal{E} \rightarrow \{1, 0\}$. $In_{A_k}(d) = 1$, if $d \in \mathcal{D}$ is an input parameter for A_k , otherwise $In_{A_k}(d) = 0$. $Out_{A_k}(e) = 1$, if $e \in \mathcal{E}$ is an output parameter for A_k , otherwise $Out_{A_k}(e) = 0$. Similarly we can define the In_{B_h} and Out_{B_h} for each action B_h in the provided protocol.

Considering the case study in Chapter 1.5 and the alignment reported in Table 3.1 and in Table 3.2, we can, for example, say that for the $< StartOrder, StartOrder.CustomerId, StartOrder.OrderId >$ of the *Blue* client, $In_{StartOrder}(StartOrder.CustomerId) = 1$, $Out_{StartOrder}(StartOrder.OrderID) = 1$, while $In_{StartOrder}$ and $Out_{StartOrder}$ assume the value 0 for any other parameter, in $\mathcal{D}$ and respectively $\mathcal{E}$.

Given the definitions of the aforementioned sets and functions we can define a parallel composition of two sequences $seq_{P_{req}}$ and $seq_{C_{prov}}$.

Definition 11 (Interaction (Parallel Composition)) *The interaction (or parallel composition) of two sequences $seq_{P_{req}}$ and $seq_{P_{prov}}$ is one of the possible $\mathcal{R} = seq_{P_{req}} \parallel seq_{P_{prov}}$, where $\mathcal{R} = R_1, R_2, \ldots, R_g, \ldots, R_z$. For each $1 \leq g \leq z$ each pair $R_g = (r_1, r_2)$, is such that:*

$$r_1 = R_g|_1 \in P_1 \cup \{\epsilon\}$$

$$r_2 = R_g|_2 \in P_2 \cup \{\epsilon\}.$$

We can call $\mathcal{R}|_1$ (respectively $\mathcal{R}|_2$) the sequence obtained by projecting $\mathcal{R}$ on the first component (respectively on the second component). Finally we can define the homomorphism $\not\epsilon$ that deletes all the instances of ϵ in the projections $\mathcal{R}|_1$ and in $\mathcal{R}|_2$, such that:

$$\not\epsilon\,(\mathcal{R}|_1) = seq_{P_{req}}$$

$$\not\epsilon\,(\mathcal{R}|_2) = seq_{P_{prov}}.$$

Referring to the theory introduced in Chapter 2, the $\mathcal{R}$ defined here, corresponds to the concept of complementary traces of Definition 8

In order to find out if the interaction between a sequence $seq_{P_{prov}}$ and a sequence $seq_{P_{req}}$ is feasible, we need to keep track of the input and output parameters exchanged. We keep track of this, introducing two sets of functions on the parallel composition $\mathcal{R}$:

$$\begin{aligned} seen_{P_{req}}(R_g, D_x) = seen_{P_{req}}(R_{g-1}, D_x) + \\ In_{R_g|_1}(D_x) - In_{R_g|_2}(D_x) \end{aligned} \tag{3.1}$$

with $seen_{P_{req}}(R_g, D_x) \in \mathbb{N}, D_x \in \mathcal{D}$ and $seen_{P_{req}}(R_0, D_x) = 0$.

[1] We here make this hypothesis only for simplyfing the presentation. This simplification will be overcome later in this section.

$$needed_{P_{req}}(R_g, E_y) = needed_{P_{req}}(R_{g-1}, E_y) + Out_{R_g|_1}(E_y) - Out_{R_g|_2}(E_y) \quad (3.2)$$

with $needed_{P_{req}}(R_g, E_y) \in \mathbb{Z}, E_y \in \mathcal{E}$ and $needed_{P_{req}}(R_0, E_y) = 0$

Intuitively the $seen_{P_{req}}$ function keeps track of how many instances of a given input parameter have been provided by actions in $seq_{P_{req}}$, invoked so far, and not yet read by actions in $seq_{P_{prov}}$. The $needed_{P_{req}}$, instead, keeps track of how many instances of a given output parameter, expected by the actions in $seq_{P_{req}}$ invoked so far, have not yet been provided by actions in $seq_{P_{prov}}$.

The $seen_{P_{req}}(R_g, D_x)$ function is used to define a *progress* property on the $\mathcal{R}$ sequence.

Definition 12 ($Progress_1$) *The g^{th} step of the sequence can be performed only if after its invocation $seen_{P_{req}}(R_g, D_x) \geq 0$ for each D_x in $\mathcal{D}$.*

The $needed_{P_{req}}(R_g, E_y)$ is used to define a *consistency* property on the $\mathcal{R}$ sequence.

Definition 13 ($Consistency_1$) *The $Consistency_2$ property holds in a sequence $\mathcal{R}$ if, after the sequence $seq_{P_{req}}$ reaches its last step, for each E_y in $\mathcal{E}$, $needed_{P_{req}}(R_g, E_y) \leq 0$.*

Intuitively an action in $seq_{P_{req}}$ expects the invoked actions on $seq_{P_{prov}}$ to return those parameters declared in the action in $seq_{P_{req}}$ description as output parameters. In order for the communication to work, $seq_{P_{prov}}$ should return at least all of the parameter instances that are required by the action in $seq_{P_{req}}$, which means that eventually at the end of $seq_{P_{prov}}$ the $needed_{P_{req}}$ functions should run to 0 for all the E_y.

In this chapter we are considering both the cases in which the invocation semantics is *synchronous* (i.e. the invoker waits for the invoked reply before proceeding), or *asynchronous* (i.e. the invoker does not need to wait for the invoked reply before proceeding). The latter case is the most general, and, is naturally modeled by the progress and consistency properties introduced previously in this section. The synchronous case needs us to introduce a *synchronous progress* notion:

Definition 14 ($SynchronousProgress_1$) *once a pair R_g having a $R_g|_1 = a_{i,g} \neq \epsilon$ appears in the current step of $\mathcal{R}$, and $seen_{P_{req}}(R_g, E_j) > 0$ for some E_y in $\mathcal{E}$, then the following steps of $\mathcal{R}$ will be $\epsilon, R_g|_2$, until a step R_{g+c} of $\mathcal{R}$ is performed, such that $needed_{P_{req}}(R_{g+c}, E_y) = 0$.*

Given the above definitions we can define our notion of feasible invocation.

Definition 15 (Feasible Interaction (simplified version)) *An interaction between a sequence of required actions $seq_{P_{req}}$ on the P_1 protocol, and $seq_{P_{prov}}$ of provided actions on the P_2 protocol, is defined feasible if, given a possible $\mathcal{R} = seq_{P_{req}} \parallel seq_{P_{prov}}$ the $Progress_1$ property holds for each step of $\mathcal{R}$ (in the case of synchronous semantics the $SynchronousProgress_1$ property, instead, holds for each step of $\mathcal{R}$). Moreover, the $Consistency_1$ property eventually holds from a given step of $\mathcal{R}$ on and the latter property holds when both P_1 and P_2 are in accepting states.*

Intuitively, the defined concept of feasible interaction mandates that, when an action of seq_{C_2} should be invoked, the input data of the former action must have been provided by some already invoked actions of seq_{C_1} (i.e. the Progress property holds). Moreover, an action of seq_{C_1} successfully finishes its computation if eventually all of its parameters are returned by some actions in seq_{C_2} (i.e. the Consistency eventually holds). This definition concretises the concept of *compatibility* introduced in Definition 9 and allows caching of input and output parameters (i.e. respectively through the values of $seen$ and $needed$) and facilitates action re-ordering.

Extensions to the concept of feasible interaction

The formalization of feasible interaction presented so far specifies only the simplest case, in which one of the networked systems involved in a $Connection$ always acts as the invoker (i.e. performs only provided actions) and the other always acts as the invoked (i.e. performs only provided actions). Normally this is not always the case, as it is shown in the case study introduced in Chapter 1.5.

In order to take into account the aforementioned situation, our formalization of feasible interaction notion needs to be extended.

Going into details, we need to consider that when examining a sequence from an Interaction, previously defined in Definition 22, now we can find also couples composed of required actions of P_{prov} and provided actions of P_{req}. To take those couples into account we have to introduce a $seen_{P_{prov}}$ and a $needed_{P_{prov}}$ functions, defined dually to $seen_{P_{req}}$ and a $needed_{P_{req}}$ (i.e. taking into account data exchanged by required actions in P_{prov} with provided actions in P_{req}). Then, similarly to properties defined in Definitions 17 and 16, we can define $Consistency_2$ and $Progress_2$ properties, and use them to refine the feasible interaction as follows:

Definition 16 ($Progress_2$) *The g^{th} step of an interaction $\mathcal{R}$ can be performed only if after its invocation $seen_{P_{prov}}(R_g, D_x) \geq 0$ for each D_x in $\mathcal{D}$.*

Definition 17 ($Consistency_2$) *The $Consistency_2$ property holds in a sequence $\mathcal{R}$ if, after the sequence $seq_{P_{prov}}$ reaches its last step, for each E_y in $\mathcal{E}$, $needed_{P_{req}}(R_g, E_y) \leq 0$.*

Definition 18 (Feasible Interaction (generalization of Definition 15)) *An interaction between a sequence of actions $seq_{P_{req}}$ on the P_1 protocol, and $seq_{P_{prov}}$ of actions on the P_2 protocol, is defined feasible if, given a possible $\mathcal{R} = seq_{P_{req}} \parallel seq_{P_{prov}}$ the $Progress_1$ and the $Progress_2$ properties holds for each step of $\mathcal{R}$ (in the case of synchronous semantics the $SynchronousProgress_1$ and $SynchronousProgress_2$ properties, instead, holds for each step of $\mathcal{R}$). Moreover, the $Consistency_1$ and the $Consistency_2$ properties eventually hold from a given step of $\mathcal{R}$ on and these properties still hold when both P_1 and P_2 are in accepting states.*

CONNECTor formalization and synthesis

Given the definition of feasible interaction, we can give a formal definition of CONNECTor. To do so, we need to introduce the notion of *counter system* and of *counter transition system* (see [41], [33] , [30]).

In practice counter transition systems are labeled transition systems, extended with the use of possibly unbounded counters. The transitions from a state to another of the system are guarded by some*Presburger arithmetic formula* [64], which can modify the value of the counters. Presburger arithmetic is a first order theory on the structure $(\mathbb{Z}, +)$, and consequently a Presburger formula is an arithmetic formula on integers, including only addition and comparison.

Definition 19 (Counter system) *A counter system $\mathcal{S}$ is a tuple (Q, n, δ), such that Q is a non-empty set of states, $n \geq 1$ is the* dimension*(i.e. the number of counters in the system which can be represented by the variables $x_1, \ldots, x_n$) and δ is the transition relation defined as a set of triples (q, Φ, q'), where q and q' are states in Q and Φ is composed of a transition label l and a Presburger arithmetic formula φ.*

For a counter system we can define the concept of a state *configuration*.

Definition 20 (Counter system state configuration) *A configuration of the counter system $\mathcal{S} = (Q, n, \delta)$ is a pair $(q, \overrightarrow{x}) \in Q \times \mathbb{Z}^n$.*

In practical terms a configuration associates each state of the counter system to a tuple of n integer values (one for each counter of the system). Given Definitions 19 and 20 we can finally define a counter transition system

Definition 21 (Counter transition system) *Given a counter system $\mathcal{S} = (Q, n, \delta)$ its transition system $T(\mathcal{S}) = (S, \rightarrow)$ is a graph such that $S = Q \times \mathbb{Z}^n$ and $\rightarrow \subseteq S \times S$ is the transition relation, defined such that $(q, \overrightarrow{x}), (q', \overrightarrow{x'}) \in \rightarrow$ if and only if there exists a transition $t \in \delta$ and $(q, \overrightarrow{x}) \xrightarrow{t} (q', \overrightarrow{x'})$. A run ρ of a transition system $\mathcal{S}$ can be defined as a non-empty sequence $\rho = (q_1, \overrightarrow{x_1}), \ldots, (q_k, \overrightarrow{x_k})$.*

Given these definitions we can now define formally a CONNECTor for two protocols P_{req} and P_{prov}.

Definition 22 (CONNECTor) *A CONNECTor for two protocols P_{req} and P_{prov} is a counter transition system $M = (S_M, \overrightarrow{M})$, defined on a counter system $\mathcal{S}_M = (Q_M, n, \delta_M)$, where Q_M is a set of tuples (a, b) such that $a \in S_{P_{req}}$ and $b \in S_{P_{prov}}$, $n = |\mathcal{D}| + |\mathcal{E}|$, and δ_M is a transition relation composed of tuples (q_M, Φ_M, q'_M). In each tuple, q_M and q'_M are states in Q_M, and Φ_M is composed of a label $l_M = (a, b)$ such that $a \in L_{P_{req}} \cup \epsilon$*

and $b \in L_{P_{prov}} \cup \epsilon$. The formula φ_M is a CLTLB(D) formula, where CLTLB(D) is a linear temporal logic defined on Presburger arithmetic (see [16] and [15] for a detailed definition of CLTLB(D)).

A run of the CONNECTor*, $\rho_M = (q_1, \overrightarrow{x_1})_M, \ldots, (q_k, \overrightarrow{x_k})_M$ is such that $q_1, \ldots, q_k$ are a feasible interaction $\mathcal{R}$ of two action sequences seq_{Req} and seq_{Prov}, on the invoker and invoked protocols respectively, while $\overrightarrow{x_c}$, with c in $[1, \ldots, k]$, represent the values of $seen$ and $needed$ functions when the* CONNECTor *reaches the state q_c. The formula in φ_M defines the increments and decrements of the values that $seen$ and $needed$ functions take in the possible state configurations following the present one, as described in formulas 3.1 and 3.2.*

The CONNECTor synthesis problem imposes some constraints on the transitions that can be realised from a given state configuration. In particular the CONNECTor synthesis problem has the purpose of ensuring that the CONNECTor synthesised allows only feasible interactions of the client and server.

Definition 23 (CONNECTor Synthesis (without considering user goals)) *Given two protocols P_{req} and P_{prov}. we say that it is possible to build a* CONNECTor *$M_{P_{req},P_{prov}}$, if there exist at least a run ρ_M representing a feasible interaction $\mathcal{R}$, between P_{req} and P_{prov}.*

CONNECTors and goals

Having defined the problem of CONNECTor synthesis we need to take into account the last of our requirements, which states that the communication between the two networked systems should satisfy some goals specified by users.

Definition 24 (CONNECTor Synthesis (considering goals)) *A connector $M_{P_{req},P_{prov}}$ can be synthesised, considering that $P_{req} \times M \times P_{prov}$ satisfies some user goals $G_1, \ldots, G_u$, written $P_{req} \times M \times P_{prov} \models \bigwedge_{x \in [1,u]} G_x$ if there exists a run ρ_m (i.e. a feasible interaction on M) in which the user goals are verified.*

The concept of verification of a CLTLB(D) formula on a counter transition system run is defined in [16], as a *control state reachability* problem on a counter transition system. We assume to express goals CLTLB(D) formulas, that can be converted into Büchi automata, as shown in [15].

A goal G is verified on a run ρ_M when, given the counter system M, resulting from the synchronised product [31] of M and of the Büchi automaton resulting from G, and its initial configuration $q_0, \overrightarrow{x_o}$, it is possible to find a finite run ρ_M that starts from $q_0, \overrightarrow{x_o}$ and reaches a final configuration of M. The finite run ρ_M is the feasible interaction satisfying G.

Obviously, a CONNECTor M can allow several feasible interaction satisfying a Goal. The number of feasible interactions contained into a CONNECTor is potentially infinite, if the protocol of at least one of the analyzed networked systems contains a loop.

For finite state automata, there exists algorithms to compute repeated reachability in presence of infinite behaviors [29]. Previous theoretical results about reachability problems on counter transition systems with infinite behaviors, proved that this problem is undecidable for that class of systems [60]. Using a model as expressive as a counter transition system is anyway desirable, since it allows us to take into account parameters caching, which in turns allows us to solve complex mismatches (see Chapter 7 for further discussion). Later advances in theory [16], proved that the reachability problems on counter transition systems with infinite behaviors can be solved considering a finite subset of that behavior. In order to check the existence of a feasible interaction satisfying a goal G on a CONNECTor M, consequently, we consider finite runs only. This gives us a partial model of the CONNECTor, but makes the problem decidable and tractable.

Abstract CONNECTor synthesis in practice

A feasible interaction satisfying G, is produced by passing as input the ontology aligned, middleware-agnostic protocols of ns_1 and ns_2, the goal G and the synchronization model explained in 3.4 to the *Zot* model checker [54]. Zot produces as output a *sequence* of actions of P_{req} and P_{prov} representing a possible feasible interaction on M, satisfying the goal G.

The produced solution, takes into account a single possible interaction between the two analyzed networked systems. In order to produce a CONNECTor that can mediate more than one possible interaction,

we collect the sequences produced by Zot, and we concatenate them. The result is a counter transition, subset of M, which can be used as a CONNECTor at run-time, to mediate the interactions of ns_1 with ns_2.

In order to produce the CONNECTor we need to complete three main steps:

- Produce a Zot sequence fi_0, representing a possible feasible interaction on M, running Zot
- Reinvoke Zot using as input: P_{prov} and P_{req} protocol models, the goal G, the synchronization model explained in 3.4 and $!fi_0$. This produces the Zot sequence fi_1.
- Repeat the second step, so that at the n^{th} repetition Zot produces the feasible interaction fi_{n-1}, using as input: ns_1 and ns_2 protocol models, the goal G, the synchronization model explained in 3.4 and $\bigwedge_{x\in[0,n-1]}!fi_x$
- The repetition ends when Zot cannot produce any other feasible interactions, or when n^{th} repetition is performed, where n is an upper bound fixed by the systems integrator.

The produced feasible interactions are then concatenated to produce a subset of M. The output of the concatenation will be used at run-time as an abstract CONNECTor, capable of mediating some interactions of P_{req} and P_{prov}.

Once the abstract connector has been produced, it is necessary to *concretize* it, realising the *mapping* described in Section 2.2.3. The implementation of this last phase is not detailed here, and we refer the reader to Chapter 4 of D1.3 [4] for an explanation about how concretization of an abstract CONNECTor happens.

3.5 Solution at Work

Let us explain on the case study in Chapter 1.5 how we can synthesize a CONNECTor. We make the hypothesis that the user goal for our case study predicates on the *Blue* client possibility of seeing the order that it is sending to the customer service closed. Examining the client protocol, in Figure 2.3, we can notice that the order is closed when the *Result* output parameter of the *CloseOrder* action is received. In that language, the informally aforementioned goal can be expressed as follows: In CONNECT we make the hypothesis that user goals are expressed in the language introduced in deliverable D1.3 [4]. The CONNECT goal language uses the most common LTL operators (i.e. [] for globaly, <> for eventually, X for next, etc.) and introduces three predicates:

- executed(c): meaning that the action represented by c has been invoked.
- received(c): meaning that the data represented by concept c have been received
- sent(c): meaning that the data represented by concept c have been sent.

The formula in the CONNECT goal language we want to specify for our example is the following:

$$<> (received(CloseOrder.Result)) \tag{3.3}$$

In Formula (3.3) the <> operator is the classical LTL $\Diamond$ operator, while $received(x)$ is a predicate indicating that the argument x is expected to be sent. Since $CloseOrder.Result$ is an output parameter, we can represent the Formula (3.3) in terms of our model, as the CLTLB(D) formula: $(<> (needed(CloseOrder.Result) = 1)||(needed(CloseOrder.Result) = -1))$.

The meaning of this formula is that the piece of data represented
by the annotation $CloseOrder.Result$ should be eventually required by P_{req}
(i.e. $(needed(CloseOrder.Result) = 1)$, in which case, according to the rules of our model, if a feasible interaction exists $CloseOrder.Result$ will be eventually provided by P_{prov}), or should be provided by P_{prov} (i.e $(needed(CloseOrder.Result) = -1)$).

This formula produces quite a simple Büchi automaton, which is depicted in Figure 3.2 According to Definition 24, in order to find a mediator satisfying the goal in (3.3), we should build the synchronised product of the mediator for those networked systems whose protocols are reported in Figure 2.3 (that we will consider as P_{req}) and in Figure 3.1 (that we will consider as P_{prov}), and the Büchi automaton in

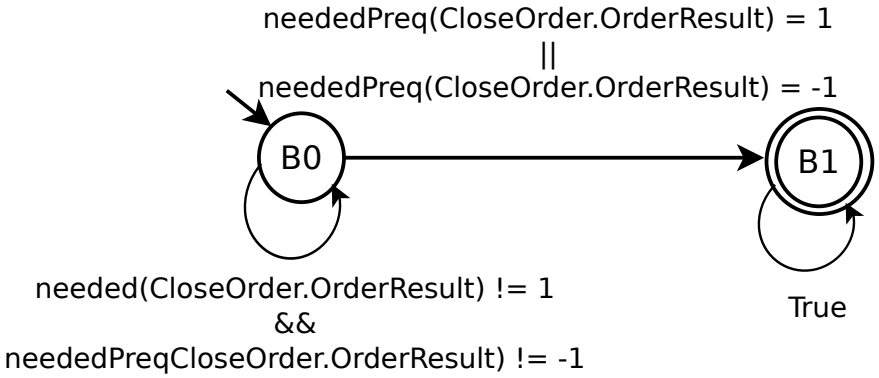

Figure 3.2: Büchi atuomaton for the goal in (3.3)

Figure 3.2, and solve, on the resulting counter transition system M, a reachability problem which can be stated as follows: is there any final state, that can be reached from the initial configuration of M, with a run ρ_M, such that the latter run is a feasible interaction. A possible run of M, realising a feasible interaction is reported in Figure 3.3. In that figure we represent the feasible interaction using the common language inferred in Table 3.1 and Table 3.2.

The run starts in the initial configuration of M, where all counters are set to 0. The first step of the run ρ_M brings M from its initial configuration, into a configuration where the counters $seen_{P_{req}}(StartOrder.CustomerID)$ and $needed_{P_{req}}(StartOrder.OrderId)$ have the value 1, through the invocation of the $StartOrder$ required action (common language action for $< StartOrder, \{CustomerId\}, \{OrderId\} >$) on the required protocol, performing no action (i.e. ϵ) on the provided protocol, and having the $needed_{P_{req}}(CloseOrder.OrderResult)! = 1$ and $needed_{P_{req}}(CloseOrder.OrderResult)! = -1$ formula, of the Büchi automaton, holding. We are here assuming an asynchronous interaction, thus in the new configuration of M the $Progress_1$ property of Definition 16 should hold. This is true since all the seen counters have a value ≥ 0.

The second step of the run brings the system into a configuration where all the $seen_{P_{req}}$ counters have the value 0, through an ϵ-move on P_{req} and the invocation of the provided action $< \overline{StartOrder}, \{StartOrder.CustomerId\}, \{\}$ (common language action for $< Login, \{CustomerId\}, \{\} >$) on P_{prov}. The invoked action takes as input an instance of $StartOrder.CustomerID$, consequently, according to the $seen_{P_{req}}$ definition we gave in Formula 3.1, the value of $seen_{P_{req}}(StartOrder.CustomerID)$ runs to 0. This meets the $Progress_1$ property. It is worth nothing that, at the same time, since all the $seen_{P_{prov}}$ counters are left set to 0, also the $Progress_2$ property holds. The executed action on P_{prov} has no output parameters, this will leave the values of all the $needed_{P_{req}}$ counters unchanged. Finally, we notice that the $CloseOrder.OrderResult$ has still not been provided by P_{req}, consequently the Büchi automaton stays in its B0 state. This latter condition will remain unchanged, until the value of the $needed_{P_{req}}(CloseOrder.OrderResult)$ counter will not change.

The third step of ρ_M brings M into a configuration S1,S2,B0, through a second $\epsilon-$ move on P_{req} and the invocation of the provided action $< \overline{StartOrder}, \{\}, \{StartOrder.OrderId, StartOrder.HasValidity\} >$ on P_{prov} (common language action for $< \overline{CreateOrder}, \{\}, \{PurchaseOrder\} >$). This action takes no input parameter and returns and instance of $StartOrder.OrderId$ and an instance of $StartOrder.HasValidity$. This makes the $needed_{P_{req}}(StartOrder.OrderID)$ function run to 0, and the $needed_{P_{req}}(StartOrder.HasValidity)$ function to run to -1. This means that an extra value of $StartOrder.HasValidity$ has been provided by $\overline{StartOrder}$ on P_{prov}. The extra value can be cached by the mediator and, if needed, returned later to P_{req}. The latter parameter is never used by P_{req}, as can be noticed from the representation of the the *Blue* client, in Figure 2.3. Consequently it will be received by the mediator and never sent to the client, and its counter will remain set to -1 for the rest of the feasible interaction. Also in this case the progress property holds, since all the $seen$ counters are set to 0.

The fourth step in the run brings M into a configuration S2,S2,B0, through the invocation of $< \overline{AddItemToOrder}, \{AddItemToOrder.OrderId, AddItemToOrder.ItemId, AddItemToOrder.ItemQuantity\}, \{\} >$ required action on P_{req} (common language ac-

tion for $< \overline{AddItemToOrder}, \{OrderId, ProductItem\}, \{\} >$). This sets the counters $seenP_{req}(AddItemToOrder.OrderId)$, $seenP_{req}(AddItemToOrder.ItemQuantity)$ and $seenP_{req}(AddItemToOrder.ItemId)$ to 1, leaving all the needed counters unvaried, since it requires no output parameters. Considering the definition of progress given in Definition 16, the current configuration of M allows the invocation of some provided actions on P_{prov} such as they take as parameters no more than an instance of AddItemToOrder.OrderId, AddItemToOrder.ItemQuantity and AddItemToOrder.ItemId. This condition is realised by the action on P_{prov}: $< \overline{AddItemToOrder}, \{AddItemToOrder.OrderId, AddItemToOrder.ItemId, AddItemToOrder.ItemQuantity\}, \{\} >$ (common language action for $< \overline{SelectItem}, \{OrderId, ItemId, ItemQuantity\}, \{\} >$). This action brings M into a state S2, S3, B0, in which all counters, but $needed_{P_{req}}(StartOrder.HasValidity)$ are set to 0.

At this point the feasible interaction continues on P_{Prov}, with the invocation of the $< ItemConfirmation, \{ItemConfirmation.OrderId, ItemConfirmation.ItemId, ItemConfirmation.ItemQuantity\}, \{\} >$ required action on P_{prov} (common language action for $< ConfirmItem, \{OrderId, ItemId, ItemQuantity\}, >$).

The invocation of this action makes the counters $seen_{P_{prov}}(ItemConfirmation.OrderId)$, $seen_{P_{prov}}(ItemConfirmation.ItemId)$ and $seen_{P_{prov}}(ItemConfirmation.ItemQuantity)$ run to 1, as previously defined in Section 3.4.

This means that the CONNECTor M receives from P_{prov} instances of $ItemConfirmation.OrderId$, $ItemConfirmation.ItemId$ and $ItemConfirmation.ItemQuantity$, and can use them, if needed, to provide input parameters of some P_{req} provided actions, or to provide output parameters for required actions of P_{prov},as stated by the definitions of $seen_{P_{prov}}$ and $needed_{P_{prov}}$ given in Section 3.4.

The feasible interaction continues with the invocation of the $< \overline{CloseOrder}, \{CloseOrder.OrderId, CloseOrder.CreditCardNumber\}, \{\} >$ required action on P_{req} (common language action for $< \overline{PlaceOrder}, \{OrderId, CreditCardNumber\}, \{\} >$, which makes also the $seen_{P_{req}}$ counters for the input parameters of this action run to 1.

P_{req}, at this point, expects to receive the confirmation for the items previously added to the order, through the $ItemConfirmation, \{ItemConfirmation.OrderId, ItemConfirmation.ItemId, ItemConfirmation.ItemQuantity\}, \{\} >$ provided action (common language action for $ReceiveConfirmation(\{OrderId, ProductItem\}, \{\})$). The three input parameters of this action will make all the $seen_{P_{prov}}$ counters run to 0, respecting the $Progress_2$ property.

In the next step of the example feasible interaction on M the $CloseOrder, \{CloseOrder.OrderId, CloseOrder.CreditCardNumber\}, \{CloseOrder.OrderResult\} >$ provided action is performed on P_{prov} (common language action for $CloseOrder, \{OrderId, CreditCardNumber\}, \{CloseOrder.OrderResult\} >$). This action makes the $seen_{P_{prov}}, ItemConfirmation.OrderId >$, $seen_{P_{prov}}, ItemConfirmation.OrderId >$ and $seen_{P_{prov}}, ItemConfirmation.OrderId >$ run to 0, this respects the $Progress_1$ property. Moreover, it makes the the counter $needed_{P_{req}}(CloseOrder.OrderResult)$ run to -1. This means that P_{prov} here sends to the mediator an instance of $CloseOrder.OrderResult$ not yet needed by P_{req}, and it also makes the Büchi automaton to go into the B1 state. It is interesting to notice that from this transition on, the Büchi automaton will continue to perform the transition labeled as “true”, since the goal presented in Formula 3.3 has been satisfied. In the reached configuration S4, SEnd, B1, the $Progress_1$ and $Progress_2$ properties hold, as well as the $Consistency_1$ and $Consistency_2$ properties, since all the $needed_{P_{req}}$ and the $needed_{P_{prov}}$ counters have a value ≤ 0. Anyway, it is not a final configuration for M, since P_{req} is not in a final state.

The latter condition is met with the last transition of the feasible interaction, the invocation of the $< CloseOrder, \{CloseOrder.OrderId\}, \{CloseOrder.OrderResult\} >$ required action on P_{req} (common language action for $< Quit, \{OrderId\}, \{result\} >$). This transition brings M in to a state in which both the progress and the consistency properties hold, and P_{req}, P_{prov} and the Büchi automaton are all in a final state. Consequently we can state that the interaction ρ_M that we illustrated is a feasible one and satisfies the goal in Formula 3.3.

3.6 Implementation Description

The Goal-based abstract CONNECTor synthesis, described in this chapter has been implemented in a Java prototype framework, represented in Figure 3.4. The current implementation of the framework does not consider the middleware abstraction phase, which we suppose is provided. Consequently it starts from the description of two middleware-agnostic networked systems and a list of goals, representing the CONNECTor intent (see Chapter 6 for details about the framework inputs).

The prototype takes as input the representations of a required and a provided affordance model protocols, P_{req} and P_{prov}, as shown in Figure 3.4.

The alphabets of the two middleware-agnostic affordance models are aligned using an *alphabet aligner* component, that performs the task through ontology matching, as described in Section 3.3. The output of this step, in the figure, is constituted of *Aligned* P_{req} and *Aligned* P_{prov}.

The aligned protocols are used as input of the *Zot input file provider*, together with the user goals. The Zot input file provider produces an input file for the Zot model checker [54]. The latter, tries to find a feasible interaction satisfying the goals, as previously specified in Section 3.4, through SMT-based bounded model checking. Zot produces as output one of the possible feasible interactions, as a *Zot trace*. The latter is a representation of the actions that the network systems should invoke and of the data they should provide, in order to communicate with each other (i.e. what is defined as a CONNECTor run in 3.4).

In order to produce a CONNECTor for more than a possible feasible interaction, the Zot trace is collected by a *Trace Collector* component, negated and passed as input to the Zot input file provider. The latter produces a new input file for Zot (as explained in Section 3.4). In this way Zot output traces are collected and concatenated into a CONNECTor. The traces generation process stops when Zot cannot find another solution for the given input, or in case an upper bound specified by the systems integrator is reached. This upper bound is needed in case the protocol of at least one of the two networked systems describes an infinite language.

3.7 Conclusion

In this chapter we introduced a possible approach implementing the CONNECTor theory. The work presented in this chapter specializes the theory presented in Chapter 2 by introducing the following hypotheses:

- The CONNECTor synthesis is driven by a goal, specified through a $Connection$. The goal expresses user's interest in making the two networked systems in the $Connection$ communicate.
- The common language identification phase, described in the theory in Section 2.2.3, is obtained through the alignment of the two networked systems alphabets, as described in Section 3.3.
- The CONNECTor synthesis happens on a per-trace base, as detailed in Section 3.4. This makes the CONNECTor able to handle complex mismatches (see Chapter 5), but makes the CONNECTor produced partial, in presence of loops in the networked systems protocols, as explained in Section 3.4.

Our contribution builds on the previous work presented in [26, 15] by adding on top of it an ontology matching process (presented in 3.3), the support for goal-based CONNECTor synthesis (see 3.4), and extending the communication model that allows us to synthesize mediators to take into account communication models different from the client-server one. The approach has been implemented in a framework that provides a specification of a possible CONNECTor between two given networked systems, given their specification and a list of connection goals. Possible future work will investigate the possibility of integrating our current work with the work carried on in CONNECT about dependability and security (see Deliverable D5.3 [11]).

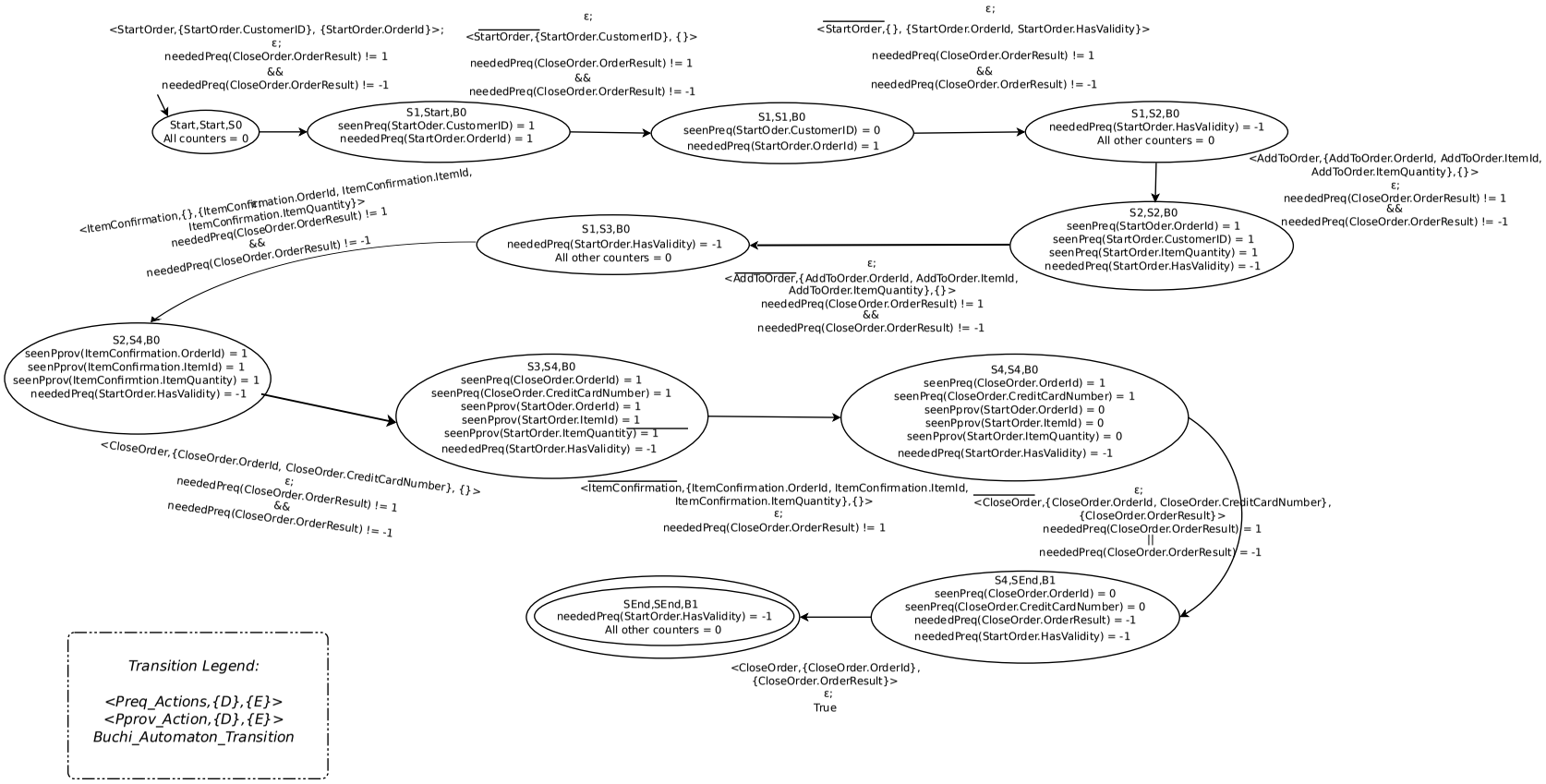

Figure 3.3: Representation of a possible run of M, for the case study in Chapter 1.5.

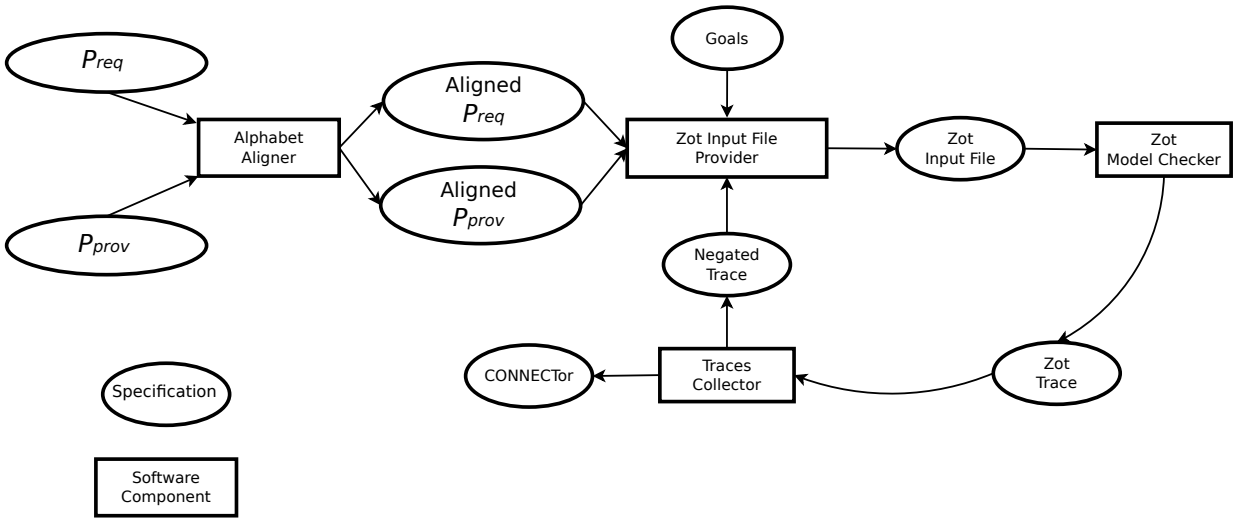

Figure 3.4: Goal-based abstract CONNECTor synthesis enabler implementation.

4 Mapping-driven Abstract CONNECTor Synthesis

In Chapter 2 we specified the main steps of the Abstract CONNECTor synthesis while in Chapter 3 we presented an implementation of this theory making two main assumptions: (i) there is no third party involved in the communication between the two considered networked systems, and (ii) the mediator does not manage the data flow within the individual systems, i.e., the mediator does not receive the output of one action and send it back to following actions of the same NS.

In this Chapter we relax this latter assumption by allowing the management of input/output flow in a sequence of actions. In particular, we infer the correspondence between the actions of the interfaces of the two NSs so as to generate the mapping processes that perform the necessary translations between actions. Various mapping relations may be defined. They primarily differ according to their complexity and inversely proportional flexibility. In Deliverable 3.2 [9], we proposed an approach for one-to-one mapping between application-specific actions according to their ontology-based semantics, i.e., there is a direct correspondence between actions based on the subsumption relation (a.k.a. is-a) defined over concepts in the corresponding ontology. In this chapter we further consider (i) one-to-many, i.e., an action is mapped to a sequence of actions; and (ii) many-to-many, i.e., a set of actions may correspond to a sequence of actions. The corresponding mappings are generated according to the mediator capabilities, which include receiving and sending messages, delaying the delivery of messages, and reasoning about the semantics of messages in order to generate messages by transforming and composing the original ones. The mediator can not create messages except for basic control messages, such as simple acknowledgments which in this case are equivalent to *owl:Nothing*. We further assume that causally independent actions of each NS are made concurrent in order to support acceptable re-ordering.

We propose an approach that combines ontology reasoning and constraint programming in order to generate a mapping between the interfaces of these NSs. Then, we use the generated interface mapping to automatically synthesize a mediator that ensures their safe interaction, i.e., it ensures deadlock-freedom and the absence of extra output actions [84].

- *Efficient interface mapping using semantic reasoning and constraint programming.* We propose an approach that combines semantic reasoning and constraint programming to identify the semantic correspondence between NSs' actions, i.e., the common language. Besides tackling the one-to-one correspondence between actions [9], we consider the more general cases of one-to-many and many-to-many mappings. Furthermore, we propose an encoding of the ontology that takes into account subsumption and the union of classes in order to make reasoning more efficient at runtime.

- *Automated synthesis of mediators to support interoperability between heterogeneous applications.* The interface mapping is performed at runtime and does not require *a priori* knowledge about the NSs; they only need to adhere to the same ontology, which is defined to reflect the terminology of the application domain. Consequently, this interface mapping may be ambiguous, i.e, one action may be semantically matched to different sequences of action from the other NSs. During the synthesis step, we explore the various possible mappings in order to produce a correct-by-construction mediator that guarantees the safe interaction of the two systems, i.e., it ensures deadlock-freedom and the absence of extra output actions in the interaction [84].

We further demonstrate the feasibility of our approach through the MICS (Mediator Synthesis to CONNECT Components) tool and evaluate its performance using various CONNECT scenarios t.

The rest of the chapter is structured as follows. Section 4.1 describes our approach to identify the common language by mapping the actions of the interfaces of the two networked systems; and how this mapping is used in the automated synthesis of mediators that promote the interaction of these systems. Section 4.2 illustrates the functioning of the approach using the Purchase Order Scenario. Section 4.3 describes the implementation of the MICS tool. Section 4.4 provides preliminary evaluation of the MICS tool. Finally, Section 4.5 concludes the chapter.

4.1 Abstract CONNECTor Synthesis

4.1.1 Support for Efficient Interface Mapping to Identify the Common Language

The first step towards the synthesis of mediators is to identify the semantic correspondence between the actions of the NSs' interfaces. In this section, we introduce a constraint-based approach to infer such correspondence by reasoning about the semantics of the interfaces' actions. We briefly introduce the principles of constraint programming and reduce the interface mapping to a Constraint Satisfaction Problem that can be efficiently dealt with using constraint programming. Finally, we propose an ontology encoding that considers subsumption and union in order to translate ontology reasoning into finite-domain constraints and speed up the reasoning at runtime.

Constraint Programming in a Nutshell

Constraint programming (CP) is the study of computational systems by stating constraints (conditions, properties) which must be satisfied by the solution [67]. Constraint programming allows to solve combinatorial problems modeled by a constraint satisfaction problem, which is formally defined as a triplet (X, D, C):

- *Variables*: $X = \{X_1, X_2, ..., X_n\}$ is the set of variables of the problem.
- *Domains*: D is a function which associates to each variable X_i its domain $D(X_i)$, i.e., the set of possible values that can be assigned to X_i.
- *Constraints*: $C = \{C_1, C_2, ..., C_m\}$ is the set of constraints. A constraint C_j is a mathematical or symbolic (global) relation defined over a subset $X^j = \left\{X_1^j, X_2^j, ..., X_{n^j}^j\right\} \subseteq X$ of variables which restricts their possible values. Constraints are used actively to deduce infeasible values and delete them from the domains of variables. This mechanism is called *constraint propagation*. Efficient algorithms specific to each constraint are used in this propagation.

Solving a constraint satisfaction problem involves finding a tuple (or tuples) $v = \left(v_1, ..., v_{n_j}\right) \in D(X)$ on the set of variables which satisfies all the constraints. It must be noted that the above definition of CP and its solving techniques is suitable for variables of finite domains. However, CP has also been successfully used for solving problems containing variables of other domains like real, rational numbers, or sets.

Using Constraint Programming for Interface Mapping

Let us consider two NSs $\mathcal{C}_1$ and $\mathcal{C}_2$ associated with interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. An interface mapping $Map_I\left(\mathcal{I}_1, \mathcal{I}_2\right)$ is a relation between an action or a sequence of actions of $\mathcal{I}_1$ and a *semantically equivalent* action or a sequence of actions of $\mathcal{I}_2$. The semantic equivalence is inferred by reasoning about the semantics of actions described in a domain ontology $\mathcal{O}$.

We reduce interface mapping to a constraint satisfaction problem and use constraint programming to solve it. Therefore, we have to: (i) specify the problem's variables, (ii) formulate the constraints over the aforementioned variables, and (iii) define a strategy to enumerate the set of solutions.

Modeling. Mapping $\mathcal{I}_1$ to $\mathcal{I}_2$ consists of finding pairs (X, Y) where $X = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ and $Y = \left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{i=1..n}$ and such that X *maps to* Y, i.e., X and Y verify some constraint C relating their actions, denoted $X \mapsto Y$. Furthermore, this pair is minimal, that is, any other pair verifying the constrain includes X and Y.

Since the interface is a finite set of actions, X (respectively Y) can be modeled through a vector that associates each action with its position in the sequence (0 indicates that the action is not included in the sequence). Therefore, many actions might have the same position if there is no dependency between their data. The interface mapping is then defined as follows:

$$
\begin{array}{ll}
Map_I\left(\mathcal{I}_1, \mathcal{I}_2\right) = & \\
\{ & (X, Y) | \\
& X = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m} \\
\wedge & Y = \left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{i=1..n} \\
\wedge & X \mapsto Y \\
\wedge & \nexists \left(X', Y'\right) | \left(X' \mapsto Y'\right) \wedge \left(X \subseteq X'\right) \wedge \left(Y \subseteq Y'\right) \\
\} &
\end{array}
$$

We distinguish between the following cases

- if $m = 1$ and $n = 1$, then it is a one-to-one mapping, denoted $X \overset{1-1}{\longmapsto} Y$, and it simply states the semantic equivalence of actions.
- if $m = 1$ and $n \geqslant 1$, then it is a one-to-many, denoted $X \overset{1-n}{\longmapsto} Y$, mapping and refers to actions split/merge, i.e., when a required action from one NS is provided by a composition of actions from the other.
- if $m \geqslant 1$ and $n \geqslant 1$, this is the most generic case and refers to many-to-many mapping, denoted $X \overset{m-n}{\longmapsto} Y$. It is used to specify the case where a composition of actions corresponds to another composition of actions.

The first case is straightforward although the concretization of the translation requires further treatment through lifting and lowering techniques. These techniques allow us to rename and restructure the data to eliminate the syntactical differences while preserving its semantics. The second case is the main focus of this chapter as it handles a large number of mismatches. Finally, we provide an initial solution for the third case, which can be easily extended in the future.

Constraints Specification. The constraints specify the requirements that the interface mapping should satisfy in order to guarantee semantic compatibility between actions. They explicitly state the semantic invariants that must be preserved by any mapping. We first give an intuitive definition in the one-to-one case, which we then extend to the one-to-many and many-to-many case.

An input action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ *maps to* an output action $\overline{\beta} = \left\langle \overline{b}, I_b, O_b \right\rangle \in \mathcal{I}_2$ written $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$, iff:

- $a \sqsubseteq b$
- $I_b \sqsubseteq I_a$
- $O_a \sqsubseteq O_b$

The intuition behind this mapping is that an input action can be mapped to an output one if the required operation is less demanding, it provides richer input data and needs less output data.

An input action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ maps to a sequence of output actions $\left\langle \overline{\beta_i} = \left\langle \overline{b_i}, I_{b_i}, O_{b_i} \right\rangle \in \mathcal{I}_2 \right\rangle_{i=1..n}$, written $\alpha \overset{1-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$, iff:

- $a \sqsubseteq \bigcup_{i=1}^{n} b_i$
- $I_{b_1} \sqsubseteq I_a$ and $I_{b_i} \sqsubseteq I_a \bigcup \left(\bigcup_{k=1}^{i-1} O_{b_k} \right)$
- $O_a \sqsubseteq \bigcup_{i=1}^{n} O_{b_i}$

The first constraint states that the operation required by α can effectively be provided through the operations b_i. The second constraint ensures that actions can only be executed if all their input data have been produced. The third one guarantees that the output produced by the execution of β_i are at least those α is expecting.

A sequence of input actions $\langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ maps to a sequence of output actions $\left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{j=1..n}$, written $\langle \alpha_1, ..., \alpha_l, ..., \alpha_m \rangle \overset{m-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$, iff:

- $\bigcup_{i=1}^{m} a_i \sqsubseteq \bigcup_{j=1}^{n} b_j$
- $I_{b_1} \sqsubseteq \bigcup_{i=1}^{l} I_{a_i}$ and $I_{b_j} \sqsubseteq \left(\bigcup_{i=1}^{l} I_{a_i}\right) \bigcup \left(\bigcup_{k=1}^{j-1} O_{b_k}\right)$
- $O_k = \emptyset\ k = 1, l-1$ and $O_k \sqsubseteq \bigcup_{j=1}^{n} O_{b_j}, k = l, m$

The constraints state that since the actions $\alpha_1, ..., \alpha_l$ do not require any output data, then progress can be made. However, once an output is required then the output actions should be executable since they have the necessary input data. They also ensure that actions can only be executed if all their input data are available. These definitions cover only a subset of all the possible many-to-many mappings. Nevertheless, the modularity of the approach allows us to add further mapping clauses without changing the overall approach.

Enumeration Strategy. As stated above, the mapping pairs (X, Y) can be modeled through a vector that associates each action with its position in the sequence. Actions on the same position do not depend on each other, and can be performed concurrently if this is allowed by the behavior of the NS. Therefore, we can easily favor solutions that increase concurrent executions by simply stating an enumeration strategy which minimizes the sum of positions.

Support for Ontology Encoding

In this section, we propose to associate numerical codes to ontology concepts in order to optimize the subsumption testing required during interface mapping. The concept hierarchy is usually represented by a directed acyclic graph where nodes are labeled with concept names and edges correspond with subsumption relationships. The hierarchy always contains a top built-in concept *owl:Thing* and a bottom concept *owl:Nothing*. Finally, the subsumption relation is both transitive and reflexive. Existing approaches to ontology encoding [38, 65] consider the subsumption relation without encoding the union construct needed for interface mapping. Therefore, we define an ontology encoding that expresses subsumption as well as union.

Subsumption reduces essentially to subset inclusion between concepts. Therefore, we define a *correct* and *complete* model[1] of an ontology $\mathcal{O}$ regarding subsumption and union using sets, which will finally be represented as bit vectors (see Algorithm 1). The algorithm initializes by assigning a unique element to each concept (Lines 1−3). Then, it propagates this elements to all sub-concepts (Lines 4−8). It then sorts the concepts declared through union axioms from the superconcept to subconcepts (Line 9). For each concept $a = \bigcup_{i=1}^{n} a_i$, it considers the set difference and splits each of its elements over the concepts a_i (Lines 12−20). It further adds these elements to all the superconcepts in order to preserve the subsumption relation(Lines 16−18). Finally, it encodes sets using bit vectors (Line 22): the size of the vectors corresponds to the number of elements of all sets; then if an element belongs to the set it will be represented by 1. Union is then represented by bitwise or as follows $Code(\bigcup_{i=1}^{n} a_i) = \bigvee_{i=1}^{n} Code(a_i)$, and subsumption tested using bitwise and $a \sqsubseteq b \Longleftrightarrow Code(a) \wedge Code(b) = Code(a)$.

4.1.2 Support for Mediator Synthesis

Given interface mappings returned by $Map_I(\mathcal{I}_1, \mathcal{I}_2)$ and $Map_I(\mathcal{I}_2, \mathcal{I}_1)$, where all input actions are involved in one mapping at least, we need to generate a mediator M that allows protocols P_1 and P_2, associated with the functionally matching NSs, to coordinate, i.e., the parallel composition $P_1 \| M \| P_2$ successfully terminates; or determines that no such mediator exists. Note that the mediator cannot generate a new functionality, it is just able to transform actions according to their semantics as specified by the interface mappings. If such a mediator exists, then we say that P_1 and P_2 are behaviorally compatible through a mediator M, written $P_1 \leftrightarrow_M P_2$.

[1] Although we do not give the proof here for conciseness and lack of space

Algorithm 1 EncodingOntology

Require: Classified ontology $\mathcal{O}$
Ensure: $Code[]$: A bit-vector associated with each class $c \in \mathcal{O}$
  for all $c \in Classes(\mathcal{O})$ **do**
    $Set[c] \leftarrow \{NewElement()\}$
  end for
  for all $c \in Classes(\mathcal{O})$ **do**
    for all $d \in Descendants(c)$ **do**
      $Set[c] \leftarrow Set[c] \cup Set[d]$
    end for
  end for
  $unionAxiomList = Sort(UnionAxioms(\mathcal{O}))$
  for all $(a = \bigcup_{i=1}^{n} a_i) \in unionAxiomList$ **do**
    $D \leftarrow Set[a] \setminus \bigcup_{i=1}^{n} Set[a_i]$
    for all $d \in D$ **do**
      for all a_i **do**
        $d_i \leftarrow \{NewElement()\}$
        $Set[a_i] \leftarrow Set[a_i] \cup \{d_i\}$
        for all $c \in Ascendants(a_i)$ **do**
          $Set[c] \leftarrow Set[c] \cup d_i$
        end for
      end for
    end for
  end for
  $SetsToBinaryVectors(Set[c], Code[c])$
  return $Code[]$

The algorithm incrementally builds a mediator M by forcing the two protocols to progress consistently so that if one requires some action(s) α, the mate must provide its (their) semantically equivalent action(s) β. Given that an interface mapping guarantees the semantic compatibility between the actions of the two NSs, then the mediator synchronizes with both protocols and compensates the differences between their actions by performing the necessary transformations. This is formally described as follows.

if $P_1 \stackrel{\alpha}{\Rightarrow} P_1'$ and $\exists P_2', \left(\alpha, \overline{\beta}\right) \in Map_I\left(\mathcal{I}_1, \mathcal{I}_2\right)$
such that $P_2 \stackrel{\overline{\beta}}{\Rightarrow} P_2'$ and $P_1' \leftrightarrow_{M'} P_2'$ (1)
then $M \stackrel{\beta}{\Rightarrow} M_t \stackrel{\overline{\alpha}}{\Rightarrow} M'$

if $P_2 \stackrel{\alpha}{\Rightarrow} P_2'$ and $\exists P_1', \left(\alpha, \overline{\beta}\right) \in Map_I\left(\mathcal{I}_2, \mathcal{I}_1\right)$
such that $P_1 \stackrel{\overline{\beta}}{\Rightarrow} P_1'$ and $P_2' \leftrightarrow_{M'} P_1'$ (2)
then $M \stackrel{\beta}{\Rightarrow} M_t \stackrel{\overline{\alpha}}{\Rightarrow} M'$

The mediator further consumes the extra output actions so as to allow protocols to progress; which is specified as follows.

if $P_1 \stackrel{\overline{\beta}}{\Rightarrow} P_1',$ and $\exists P_2$ such that $P_1' \leftrightarrow_{M'} P_2$
then $M \stackrel{\beta}{\Rightarrow} M'$ (3)

if $P_2 \stackrel{\overline{\beta}}{\Rightarrow} P_2',$ and $\exists P_1$ such that $P_2' \leftrightarrow_{M'} P_1$
then $M \stackrel{\beta}{\Rightarrow} M'$ (4)

Finally, when both protocols reach a terminating states, then the mediator also terminates.
END $\leftrightarrow_{\mathsf{END}}$ END (5)

Note that the interface mapping is not necessarily a function, since an action (or a sequence of actions) may be mapped to multiple actions (or sequences of actions). The synthesis algorithm (see Algorithm 2) has to deal with this ambiguity. The algorithm selects the enabled mappings according to the actions triggered in the other protocol (Line 5). Then, it non-deterministically chooses one of them, and verifies that the selected mapping rule will lead to a successful termination using Algorithm 3 (Line 6) and compose

these rules in parallel (Line 10). If there is no triggerable action, then Algorithm 2 fails (Lines 7-9). An instance successfully terminates when it reaches an END state (Lines1- 2).

Each instance of Algorithm 3 maintains a link to all the mappings that have been applied (Line 4). An instance fails if there is no possible mapping rule that can be applied, otherwise it returns a link to the sequence of rules that have been applied in order to reach a terminating state.

Algorithm 2 FindMediator

Require: P_1, P_2
Ensure: M
1: **if** $P_1 = END$ and $P_2 = END$ **then**
2: **return** END
3: **end if**
4: $M \leftarrow END$
5: **for all** $(\alpha, P_i') \mid P_i \xrightarrow{a} P_i'$ and $\exists \overline{\beta}, (\alpha, \overline{\beta}) \in Map_I(I_i, I_{3-i})$, $i \in \{1, 2\}$ **do**
6: $M' \leftarrow$FindMappingTrace$(i, \alpha, P_i', P_{3-i})$
7: **if** $M' =$ **fail then**
8: **return fail**
9: **end if**
10: $M \leftarrow M \| M'$
11: **end for**
12: **return** M

Algorithm 3 FindMappingTrace

Require: i, α, P_i', P_{3-i}
Ensure: M
1: **for all** $(\overline{\beta}, P_i') \mid P_{3-i} \stackrel{\overline{\beta}}{\Rightarrow} P_{3-i}'$ and $(\alpha, \overline{\beta}) \in Map_I(I_i, I_{3-i})$ **do**
2: $M' \leftarrow$FindMediator(P_1', P_2')
3: **if** $M' \neq$ **fail then**
4: **return** $\alpha \rightarrow \overline{\beta} \rightarrow M'$
5: **end if**
6: **end for**
7: **return fail**

Theorem 1 *if $P_1 \leftrightarrow_M P_2$ then $P_1 \| M$ and P_2 are in total matching. Each transition α such that $P_1 \stackrel{\alpha}{\Rightarrow} P_1'$ is associated with some transition $M \stackrel{\overline{\alpha}}{\Rightarrow} M_t$ (Statement (1)), and each transition $\overline{\beta}$ such that $P_1 \stackrel{\overline{\beta}}{\Rightarrow} P_1'$ synchronizes either with (i) $M_t \stackrel{\beta}{\Rightarrow} M'$ if there exists a mapping involving it (Statement (2)), or (ii) with $M \stackrel{\beta}{\Rightarrow} M'$ (Statement (3)). Consequently, $\alpha(P_1 \| M) = \alpha P_1 \setminus \alpha M = \alpha P_2$ (where $\setminus$ refers to set difference). In the first case, $P_1 \| M$ can perform $\overline{\beta}$, which is guaranteed to be enabled by construction P_2. In the second case, this means that an action α was required by P_2. Finally, if an action $\overline{\beta}$ was enabled by P_2, then M will necessary perform the dual action β.*

Theorem 2 *if $P_1 \leftrightarrow_M P_2$ then the parallel composition $P_1 \| M \| P_2$ is deadlock free. By construction, the mediator is synthesized only if both P_1 and P_2 reach an END state. Furthermore, the mediator absorbs all extra output actions so as to allow the processes to progress.*

4.2 Solution at Work

As mentioned in the introduction of this chapter, our approach does not consider third-party interactions and assumes that causally-independent actions are made concurrent. Therefore, we take variants of the Moon and Blue systems by (i) abstracting communications with the payment system, and (ii) putting independent actions in concurrence.

Note also that, the Blue specification does not guarantee that the number of loops on *AddItemToOrder* and that of *ConfirmItem* are the same. Since communication is initiated by input actions, then the number of iteration on item confirmation should be the one of Blue.

Consequently, the Blue specification becomes as the following:

```
1  Blue  =  (<StartOrder, {CustomerID}, {OrderID}>→ P1),
2  P1    =  (<AddItemToOrder,{OrderID, ProductItem}, ∅>→ P1
3           | <PlaceOrder,{OrderID, CreditCardNumber}, ∅>→<Quit,{OrderID},{OrderResult}> P2),
4  P2    =  (<GetConfirmItem,{orderID, productItem}, ∅>→ P2| terminate→ END).
```

The specification of the Moon service is as follows:

```
1  Moon  =  (<Login, {CustomerID}, {SessionID}>→<CreateOrder, {SessionID}, {PurchaseOrder}>→ P1).
2  P1    =  (<SelectItem, {OrderID, ItemID}, ∅>→<SetItemQuantity, {OrderID, ItemID, Quantity}, ∅>
3           | <CloseOrder, {OrderID, CreditCardNumber},{OrderResult}>→ P2),
4  P2    =  (<ConfirmItem, {OrderID, ItemID, Quantity}, ∅>→ P2| terminate→ END).
```

Interface Mapping. The first solution given by the constraint solver for the purchase order systems is:

$$Map_I\left(\mathcal{I}_{Blue}, \mathcal{I}_{Moon}\right) = \{(<1,0,0,0,0>, <1,2,0,0,0,0>), (<0,1,0,0,0>, <0,0,1,1,0,0>), (<0,0,1,1,0>, <0,0,0,0,0,1>)\}$$

which corresponds to:

$Map_I\left(\mathcal{I}_{Blue}, \mathcal{I}_{Moon}\right) = \{$
< StartOrder, {CustomerID}, {OrderID} > ↦
  << Login, {CustomerID}, {SessionID}>, < CreateOrder, {SessionID}, {PurchaseOrder}>>
< AddItemToOrder,{OrderID, ProductItem}, ∅ > ↦
  << SelectItem, {OrderID, ItemID}, ∅>→<SetItemQuantity, {OrderID, ItemID, ItemQuantity}, ∅>>
<< PlaceOrder,{CustomerID, OrderID}, ∅>, < Quit,{OrderID}, {OrderResult}>> ↦
  < CloseOrder, {OrderID}, {OrderResult}>
}

Consider for example the first mapping; it specifies that the *StartOrder* input action is equivalent to the sequence of *Login* and *CreateOrder* output actions. Indeed, as specified by the purchase ontology (see Figure 2.2) the *CreateOrder* concept subsumes the *StartOrder* concept. However, the *CreateOrder* operation requires a *SessionID* as an input concept while *StartOrder* only provides the *CustomerID*. Nevertheless, when composed with *Login*, the initial subsumption still hold, i.e., *StartOrder* $\sqsubseteq$ *Login* $\bigcup$ *CreateOrder*. In addition, the input required by *Login*, i.e., *CustomerID*, is provided by the *StartOrder* and the input of *CreateOrder*, i.e., *SessionID*, is provided as an output by the previous action, i.e., *Login*.

$$Map_I\left(\mathcal{I}_{Moon}, \mathcal{I}_{Blue}\right) = \{(<0,0,0,0,1,0>, <0,0,0,0,1>)\}$$

which corresponds to:

$Map_I\left(\mathcal{I}_{Moon}, \mathcal{I}_{Blue}\right) = \{$
< ConfirmItem,{OrderID, ItemID, ItemQuantity}, ∅ > ↦
  <GetConfirmation, {OrderID, ProductItem}, ∅>
}

Mediator Synthesis. The interface mapping is used to synthesize the the following mediator:

1 M1 = (<Login, {CustomerID}, {SessionID}>→<CreateOrder, {SessionID}, {PurchaseOrder}>
2 →< $\overline{\text{StartOrder}}$, {CustomerID}, {OrderID} >→ M1| terminate → END).
3 M2 = (<SelectItem, {OrderID, ItemID}, ∅>→<SetItemQuantity, {OrderID, ItemID, ItemQuantity}, ∅>
4 →< $\overline{\text{AddItemToOrder}}$,{OrderID, ProductItem}, ∅ >→ M2| terminate → END).
5 M3 = (< CloseOrder, {OrderID}, {OrderResult}>→< $\overline{\text{PlaceOrder}}$,{CustomerID, OrderID}, ∅>
6 →< $\overline{\text{Quit}}$,{OrderID}, {OrderResult}>→ M3| terminate → END).
7 M4 = (<GetConfirmation, {OrderID, ProductItem}, ∅>
8 →< $\overline{\text{ConfirmItem}}$,{OrderID, ItemID, ItemQuantity}, ∅ >→ M4| terminate → END).
9 ||M = (M1||M2||M3||M4).

4.3 Implementation

As part of the CONNECT synthesis enabler, we implemented the MICS (Mediator synthesIs to CONNECT componentS)[2] tool to automatically generate the mediator model. This implementation is made up of three modules: (i) The ontology-encoding module, (ii) The interface-mapping module, and (iii) The mediator-synthesis module.

The ontology-encoding module (see Figure 4.1-❶) associates numerical codes to ontology concepts in order to optimize the subsumption testing required during interface mapping. It further takes into account union of concepts. It uses Pellet reasoner[3] to classify the ontology. Pellet is an open-source java library for OWL DL reasoning.

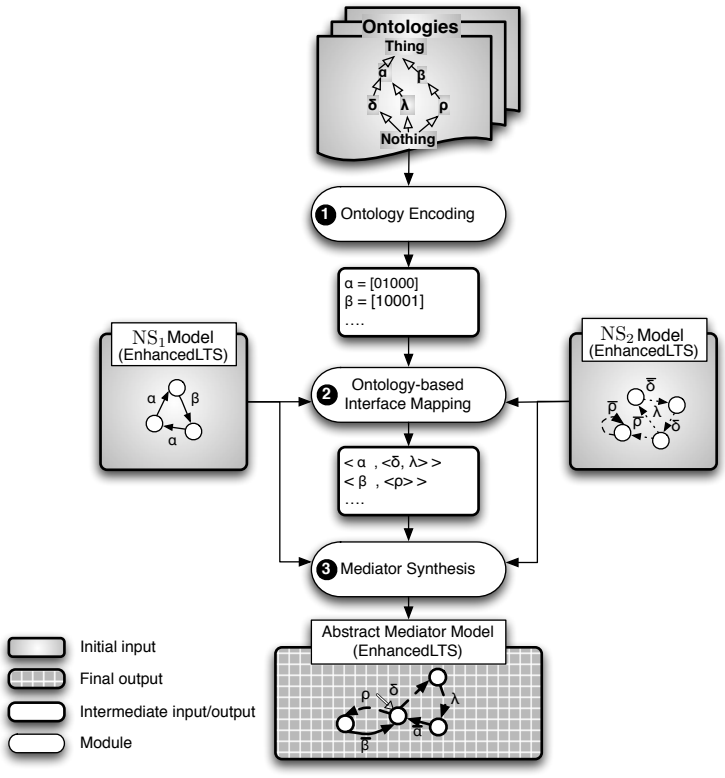

Figure 4.1: Overview of the abstract architecture of the MICS tool

[2] http://www-roc.inria.fr/arles/software/mics/
[3] http://clarkparsia.com/pellet/

The interface-mapping module (see Figure 4.1-❷) identifies the semantic correspondence between the actions of the NSs' interfaces. We formalize interface mapping as a constraint satisfaction problem for which we use Choco solver[4] to find all the possible solutions. Choco is an open-source java library for constraints solving and constraint programming. It is built on an event-based propagation mechanism with backtrackable structures.

The mediator-synthesis module (see Figure 4.1-❸) relies on these mappings to generate the mediator. In a first step, we generate the parallel composition of the mapping processes and verify that the overall system successfully terminates using the LTSA[5] (Labeled Transition System Analyzer) model checker. LTSA is a free Java-based verification tool that automatically composes, analyzes, graphically animates FSP processes and checks safety and liveness properties against them. In a second step, we are improving the algorithm so as to deal with ambiguous mappings, i.e., when an action from one NS may semantically be mapped to different actions from the other.

4.4 Evaluation

There is a lack of testbeds for extensive testing of mediation solutions. Therefore, to provide initial insight into the benefits of using our approach to support interoperability between heterogeneous systems, we used several scenarios described within the CONNECT project. They define various interoperability problems with both process and data heterogeneities. In particular, we consider the example of heterogeneous instant messaging applications [3], which represent typical one-to-one mismatches. We also experimented the tool with the photo sharing scenario presented in [45]. Besides the one-to-one mismatches, this scenario shows combined middleware and application heterogeneity–although we focused on the abstract synthesis in this evaluation. We considered the travel agency scenario originally described in [18]. This scenario goes one step further by introducing one-to-many mismatches. Finally, we considered the purchase order scenario used all through this deliverable. Note however that we didn't consider the re-ordering mismatch as putting causally-independent actions in concurrency in not implemented by our tool for the moment.

We measured the time for each step to synthesize the mediator on a Mac computer with a 2,7 GHz processor and 8 GB of memory.

Table 4.1 gives a succinct description of each scenario and shows the processing time (in milliseconds) for each step of the approach.

Scenario	Mismatches	Encoding	Mapping	Synthesis
Purchase Order	one-to-many, reordering and loops	1476	41	<1
Travel Agency [18]	one-to-many and combined mdw-app heterogeneity	1512	16	<1
Photo Sharing [45]	one-to-one and combined mdw-app heterogeneity	2652	37	<1
Instant Messaging [3]	one-to-one	300	6	<1

Table 4.1: Processing time (in milliseconds) for each step of mediation

One can note that the encoding phase is the most time consuming step as it necessitates reasoning about the ontology and inferring the hierarchy of concepts. This time depends on the size of the ontology. Nevertheless, existing ontology reasoners can efficiently handle very large ontologies. Pellet for example classifies the NASA SWEET ontology (up to 6000 concepts) in less than 8s [68]. Since the interface mapping and the subsequent synthesis rely on this encoding to test subsumption, which amounts to bit-vector operations, their execution times are much less. Moreover, one can see that there is no correlation between the time for encoding and the time for interface mapping. Indeed, the size of the ontology impacts the encoding but not the synthesis.

[4] http://choco.emn.fr/
[5] http://www.doc.ic.ac.uk/ltsa/

4.5 Conclusion

In this chapter we presented an approach to infer mappings between actions by reasoning about the semantics of their data and operations. These mappings guarantee the substitutability of the data input/output of the actions in order to allow their safe translation. We then use these mappings to automatically synthesize a correct-by-construction mediator.

While we focus in this deliverable on the automated synthesis of abstract CONNECTors models (a.k.a. mediators) as eLTSs, they need to be refined into *concrete* models, that is, a CONNECTor, and deployed atop of Starling for actual interoperability achievement as presented in Deliverable D1.3 [4]. In particular, a CONNECTor (i) intercepts the input messages of one networked system, (ii) parses them in order to obtain an action, (iii) executes the mapping rules to translate the actions of one systems into actions of the other, and (iv) forwards them to the other networked system.

An important aspect of our approach is that the mediator is made up of a set of mapping rules, which facilitate integration with Starlink, which is the execution engine of CONNECTors [4].

5 Assessment

The approaches presented respectively in Chapter 3 and Chapter 4, tackle the problem of instantiating the CONNECTor theory introduced in Chapter 2 making different hypotheses, that make each of the two approaches more suitable for a particular situation. The approach presented in Chapter 3 is based on the hypotheses that the identification of the common language takes place as alphabets alignment, that the CONNECTor is built through model checking on the networked systems protocols, whose alphabets have been aligned, that the solution is synthesized on a trace-wise base and that the synthesis process is driven by a user goal. The approach presented in Chapter 4, instead, assumes that the identification of the common language takes place as the identification of some mapping rules, that are fed to a reasoning mechanism, which produces a solution checked through a model-checker to prove its correctness, and that a solution should be produced for the whole protocols.

In this chapter we first assess the implementations with respect to each other, by comparing them on the base of the mismatch classification present in [71] (see Section 5.1), then we position our work with respect to relevant related work (see Section 5.2).

5.1 Comparing Goal-based and Mapping-driven Abstract CONNECTor Synthesis

The goal-based approach relies on an alphabet alignment process for the common language identification. The alignment substitutes the alphabets of the two networked systems, which the CONNECTor should be built for, with a common alphabet. The aligned networked systems protocols are then fed as input to a model checker, which searches one or more possible *feasible interactions*, using the user goal to reduce the search space. The mapping-based approach, instead, implements the common language identification as a set of mapping rules. These mapping rules are then applied to the product of the networked systems protocols and a model checker is used to determine if a CONNECTor exists between the two systems.

As it can be noticed from previous chapters, the different hypotheses the two approaches make are strongly correlated to the way the abstract CONNECTor is generated. In the goal-based approach, the CONNECTor is synthesized through the solution of a reachability problem on the product of the two networked systems protocols. In order to solve the reachability problem, it is necessary that the two systems have the same alphabet. The common alphabet is obtained by projecting the NS alphabets on least common concepts. Moreover, the solution in this case is produced one trace at a time. In order to be sure to capture only those traces in the protocol product which are useful to the user, the goal-based synthesis uses a goal to limit the search space.

The mapping-based implementation produces a set of mapping rules during the common language identification phase. These mapping rules not only consider the data flow across the communicating systems but also manages the input/output flow with individual systems hence injecting the outputs of one action as inputs of following actions. Furthermore, the mapping rules guarantee the substitutability of data by verifying that subsumption holds between input/output data. Then, it checks total behavioral matching between the mediated systems, i.e., the parallel composition of the system and the associated mapping processes. This solution allows the mapping-based approach to find a solution for the entire protocol, if this exists.

However, mapping-driven synthesis succeeds only if each input action has all its outputs available at time of occurrence, which is essential to prove the correctness of the mediator. Hence, many-to-many mismatches with asynchronous semantics and loops cannot be handled effectively. The Goal-based approach, instead, bases its matching phase on a reachability problem on a counter transition system. This approach allows to consider both synchronous and asynchronous behaviors, loops and allows to solve merge actions and re-ordering mismatches. On the other hand,The Goal-based approach is able to produce only subsets of the whole CONNECTor in those cases in which the interactions allowed by the networked systems protocols are infinite.

Table 5.1 evaluates the goal-based and mapping-driven abstract synthesis approaches against common mismatches [71]:

- *Signature mismatch:* concerns actions with different naming, which is naturally treated in both approaches by considering the semantics of actions .
- *Splitting of actions:* relates to having an input action of one system realized by a number of output 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.
- *Merging of actions:* defines an output action of one system that realizes a number of input actions of the other. The goal-based synthesis solves it if the input actions do not require any return parameter, or use asynchronous invocations. This is necessary because, with a synchronous semantics, an action of the output sequence would be blocked waiting for its output, that in general will not come until the required action is performed. The mapping-driven synthesis handles it only in the case the input action do not require output data.
- *Extra send (or missing receive):* During the synthesis of the mediator, both approaches ensure that any extra parameters, sent and not freseen by the second party in communication, is consumed in order to avoid deadlock.
- *Extra receive (or missing send):* a required action on one of the systems expects some input parameters are not provided by the other party involved in the communication. Both approaches do not handle this mismatche as it violates the definition of Compatibility (see Definition 9). In general, a solution to this kind of mismatches can be found if the missing parameters have default values that can be used to hide their absence. Anyway the current implementations proposed in this deliverable do not consider this case.
- *Ordering mismatch:* This concerns the re-ordering of actions so that networked systems may indeed coordinate. While the mapping-driven synthesis does not handle this case unless the networked systems models are made concurrent *a priori*. The goal-based one is able to manage this mismatch assuming asynchronous semantics.

	Goal-based Synthesis	Mapping-driven Synthesis
Goals	Yes	No
Common Language	Alphabets Alignment	Mapping Rules
Mediator Synthesis	Per trace	Whole protocol
Signature mismatch	Yes	Yes
Splitting of actions	Yes	Yes
Merging of actions	Yes, asynchronous semantics needed	Only in the case input actions do not require output parameters
Extra output	Yes	Yes
Extra input	No	No
Ordering mismatch	Yes, if there is no extra output. Needs asynchronous semantics.	No
Complete CONNECTor produced	No, if eLTSs have loops	Yes

Table 5.1: Comparing goal-based and mapping-driven abstract CONNECTor synthesis

5.2 Assessing against Related Work

Interoperability and mediation are very popular topics in the literature and have been investigated in several contexts. Indeed, since the early days of networking, many efforts have been done in several directions including for example formal approaches to protocol conversion [24, 52, 63], and their extension towards reducing the algorithmic complexity of protocol conversion [51]. Interoperability and mediation of protocols have received attention in many fields among which: integration of heterogeneous data sources [82, 81], software architectures [37], architectural patterns [23], design patterns [36], patterns of connectors [80, 73], Web services [14, 28, 75, 53, 47], and algebra to solve mismatches [32] to mention few.

Many approaches support networked system interoperability using mediators [45]. Unlike most approaches in literature, in this deliverable we considered a communication model that goes beyond the client/server one. The work we described in this deliverable is related mainly to the *Synthesis of* CONNECT*ors* and to the *Middleware Interoperability*.

5.2.1 Synthesis of CONNECTors

The approaches to the synthesis of CONNECTors present in literature, can be categorized into those that require human intervention in the definition of a mediator, and those that offer some automatic tool.

Manual or Semi-automated Approaches

Among the manual approaches, we mention here the ones in [14] and in [32] which offer a methodology for a developer to manually develop mediators, and those in [34] and [35] which offer a model checking based approach to detect networked system conversational protocol inconsistencies, but require a developer to solve them.

Approaches that propose a semi-automatic methodology to synthesize CONNECTors feature Spitznagel and Garlan [74], and, more recently, Chang *et al.* [27]. They propose to define some mismatches and their solution at design time and a runtime framework that is capable of combining the mismatches solution in order to solve more complex mismatches.

Other solutions [40] and in [61] propose a tool to assist humans in mediator development by providing hints about possible mismatches and suggesting possible solutions. The suggestions are produced by comparing the XML schemas of individual messages while taking into account their directions (in/out) and considering the associated behavioral protocol. Then, they simulate the interaction protocol in order to filter out the plausible mappings and requires the user to check the conflicting mappings. All of the approaches, require still a substantial intervention of a human system integrator. The CONNECT project aims at reducing this intervention and possibly make the CONNECTor generation process totally automatic. For this reason the work presented in this deliverable goes beyond the manual or semi-automated approaches, proposing a theory and two of its possible implementations that automate completely the synthesis process.

Automated Approaches

Automated approaches fall into two main categories: generative approaches, that try to synthesize a mediator inferring it from the specifications associated to networked systems, and restrictive approaches, that try to restrict the networked system behaviors in such a way that mismatches are avoided. With respect to this distinction, our approach can be considered generative, since it generates a CONNECTor which mediates the interactions of the analyzed networked system.

In [25] a restrictive approach which proposes a formal framework and a tool to build mediators is devised. This approach requires an *adaptation contracts* (i.e. mappings between names of different actions) to be provided and translates the conversational protocols of networked systems involved in the mediator construction into a Petri net model. Using this model the approach can handle complex cases of protocol mismatches. The approach in [44] aims at creating mediators by enforcing some desired properties out of a set of behaviors exhibited by the networked systems. This approach starts from the behavior of networked systems to adapt and from the property to enforce, specified as a Message Sequence Chart [46] and, through model checking techniques, generates some code cutting off those behaviors not verifying the desired property. This work was also extended in [76] to enable mediators to enforce some QoS constraints. The aforementioned solution are different from ours because they consider that the networked system protocol is composed by a set of messages, but they neglect the data-flow constraints, while our approach considers them.

In generative approaches to CONNECTors synthesis in literature we distinguish two phases that should be present: the common language identification phase, and a protocol matching phase. Not all the approaches support both phases. In particular the early works in the field assume the common language synthesis phase as given. Among these approaches, pioneer work by Lam [52] uses *image protocols* to reason about the existence of a CONNECTor. An image protocol is derived from a given protocol by

partitioning its state set. Two protocols are then compatible if they can be projected onto a common image protocol. However, the image protocol should be specified using an intuitive understanding of the protocols. In their seminal paper, Yellin and Strom [84] propose an algorithm for automated synthesis of CONNECTors based on unambiguous predefined interface mappings. An example of generative approach based on a subset of π-calculus is introduced in [19]. The approach can deal with complex types of mismatches but it requires a developer to provide an alignment between messages and data exchanged by networked systems if they have different signatures. In our work we consider that both the common language synthesis and the matching phases should be automated.

In some other automated approaches in literature, no rigorous protocol matching phase is supported. For instance, WSMO [28] defines a formal description language that integrates ontologies with state machines for representing Semantic Web Services. It also proposes a runtime mediation framework, the Web Service Execution Environment (WSMX), which mediates interaction between heterogeneous services by inspecting their individual protocols and performs the necessary translation on basis of pre-defined mediation patterns. However, the composition of these patterns is not considered, and there is no guarantee that it will not lead to a deadlock. Vaculín *et al.* [77] devise a mediation approach for OWL-S processes. They first generate all requesters paths, then find the appropriate mapping for each path by simulating the provider process. This approach deals only with client/server interactions and is not able to generate a CONNECTor if many mappings exist for the same action. Finally, Wu *et al.* [83] present an automated approach to process mediation taking into account the semantics of services in terms of their input/output data and preconditions and effects. The approach is based upon planning and requires predefined patterns to manage some process constructs such as choice or loops.

Nevertheless, there is a growing interest towards having automated generative approaches that consider both the common language synthesis and the protocol matching phases, and that are able to handle complex mismatches. Approaches that develop mediators only for part of the networked systems' behaviors are presented in [20] and in [56]. The latter was implemented in an open source tool.[1]. While both these approaches appear to fulfill our need for supporting interaction protocol mapping, they may present some shortcoming in terms of performances due to the high cost of exhaustive graph exploration algorithms that could prevent their usage in on-the-fly mapping derivation. While no data about performances are available for the approach in [20], some previous work of the authors presented in [26], exploited the tool offered by [56] and showed that the mapping scripts building time required by [56] tool is remarkably higher than the one obtained exploiting the approach in Chapter 3.

To the best of our knowledge the only approach capable to synthesize a mediator for two networked systems considering also the goal that their communication should satisfy is the one in [17]. With respect to this work, ours is different since it models the networked systems and their communication in a more expressive and flexible way. This allows the approach in in Chapter 3 handle reordering mismatches, that in [17] are not considered. Finally the work presented in this deliverable is different from the other presented in literature, since the previous approach focus on building a mediator specifically for client-server interactions, as this form of communication is typical in service oriented systems. Here we exploit the middleware communication model proposed in last year's deliverable D3.2 [9], in order to take into account also other types of middleware. This model extension constitutes an innovation element with respect to the rest of the existing literature, as, to the best of our knowledge, all the cited works consider only client-server interactions.

5.2.2 Middleware Interoperability

Middleware stands as a conceptual paradigm to connect applications effectively despite heterogeneities in the underlying hardware and software. Nevertheless, each middleware defines a specific message format and coordination model making it difficult (or even impossible) for applications using different middleware to interoperate. Therefore, solutions that bridge applications across different middleware systems have been devised. Enterprise Service Buses [57] promote loose coupling by implementing protocol translation through a common intermediary protocol, while interoperability platforms enable protocol substitution at runtime by exploiting reflection [39]. However, these solutions require bridges/adaptors to be created beforehand, which necessitate a substantial development effort and considerable knowledge about

[1] http://sourceforge.net/projects/dinapter

the application-domain. Therefore, solutions that reduce the effort of developing bridging solutions have emerged [45]. Specifically, z2z [22] proposes a domain-specific language to describe the protocols to be made interoperable as well as the translation logic to compose them and then generates the corresponding bridge. However, this solution requires the developer to specify the translation to be made and hence to know both protocols in advance whereas in our approach, each protocol is independently specified and the translation is produced automatically. Furthermore, the abstraction and concretization are performed at runtime while the binding to specific middleware instances is done by Starlink which takes care of the middleware specificities, instantiates the appropriate parsers and composers and deploys the concrete CONNECTor in the environment [4].

6 Abstract CONNECTor synthesis enabler

In Chapter 2 we updated the CONNECTor theory, originally proposed in [42], while in Chapter 3 and 4 we presented two possible algorithm to implement the CONNECTor theory.

In this chapter we describe a concrete architecture that allows us to unify the two aforementioned approaches into a common framework, the *Synthesis Enabler*. In Section 6.1, we describe the interface that the Synthesis Enabler exhibits to the other CONNECT enablers while in Section 6.2 we present its internal architecture, which includes the implementations described in Chapter 3 and in Chapter 4.

Figure 6.1 outlines the architecture of the synthesis enabler. The Discovery enabler calls the `synthesiseAbstractMediator` method in order to initiate the synthesis process after finding a functionally compatible pair of NSs. The Synthesis enabler considers the models (eLTSs) of these networked systems and generate a CONNECTor model that allow them to interoperate. Then, it verifies that this CONNECTor model meets the non-functional requirements of the interaction between the two NSs by calling the `Analyze` method as provided by the Connectability enabler. Finally, it sends this model to the Deployment enabler in order to be executed on top of the Starlink framework [4].

While the synthesis enabler displays a unique interface, there are two approaches to the synthesis of mediators that are implemented: goal-based and mapping-driven synthesis. The goal-based synthesis maps the sets of actions (alphabets) of both systems into a common one and use it to re-write their models, i.e., the alphabet aligner. Then it encodes both models and the goal as a satisfaction problem and solves is using an SMT model checker. The model checker gives only one feasible interaction, therefore this process is repeated until all the feasible interactions are found. The mapping-driven synthesis encode the ontology and uses this encoding to formalize a constraint satisfaction problem between the interfaces (alphabets) of the NSs. It uses then this mapping to construct a correct-by-construction mediator that enable the interoperation of the two systems. Hence, the constraints between actions and their associated input/output data are verified during the interface mapping and not during the model checking phase.

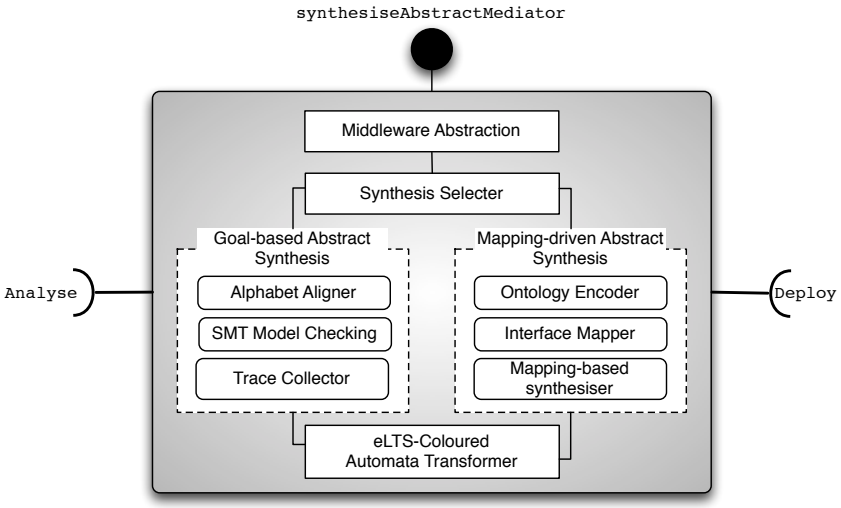

Figure 6.1: Overview of the architecture of the synthesis enabler

Both approaches relies on ontologies to reason about the semantics of actions and generate the mediator allowing heterogeneous NSs to interact properly. However, there are many differences that make each approach more suitable for specific cases. First, considering the goal allows the synthesis approach to mediate only partial matching between the behavior of the NSs. However, when such a goal is not provided, this approach needs to test all the possible traces and collect them by the end. Therefore, the mapping-driven synthesis is well suited for the cases where the goal is not specified as it considers the whole behavior of systems and checks that each possible execution of one system can possibly be mapped to an execution of the other system. Moreover, the mapping-driven synthesis can detect the impossibility of mediation while building the ontology-based action mapping thanks to the constraints

that these mappings need to satisfy, and in this case avoid expensive behavioral checking. However, mapping-driven synthesis succeed only if each input action has all its outputs available at time of occurrence, which is essential to prove the correctness of the mediator. Hence, many-to-many mismatches with asynchronous semantics cannot be handled, whereas the goal-driven synthesis manages both synchronous and asynchronous semantics for actions.

Both approaches relies on the same middleware abstraction and to transform the mediator model into a k-coloured automaton, which can be executed by Starlink. A k-coloured automaton [21] is a refinement of the mediator's eLTS. In particular, it associates each state with a marking (aka colour) to differentiate between the communication with each networked system. Furthermore, the actions are refined so as to reflect the specificities of the middleware used by each system.

In the following sections, we define both the external API of the synthesis Enabler and its internal architecture.

6.1 Synthesis Enabler API

The method to be invoked in order to synthesize the mediator is the `synthesiseAbstractMediator` of the interface `Syhthesise`:

```
1  public interface synthesise{
2   /**
3   * Generates the mediator models, sends it to the deployment enabler.
4   * @param The eLTS representation of NS1
5   * @param The eLTS of NS2
6   * @param A list of LTL formulas that specifies the goals that the mediator should verify
7   * @return The The XML model of the mediator between NS1 and NS2
8   * @throws if fails to generate the mediator
9   */
10  public ColouredAutomaton synthesiseAbstractMediator(EnhancedLTS ns1, EnhancedLTS ns2,
        List<LTLFormula> goal)
11  throws  NoMediatorException;
12 }
```

The `EnhancedLTS` and the `ColouredAutomaton` are the Java objects implementing the eLTS and the k-coloured automaton respectively. If the synthesis is unable to generate the mediator or if the Connectability analysis reveals possible failures regarding non-functional properties, then the enabler raises a `NoMediatorException` exception.

6.2 Synthesis Enabler Internal Architecture

The internal architecture of the Synthesis Enabler follows the conceptual view of the abstract CONNECTor synthesis presented in Figure 2.1. Based on that model, we propose a facade extensible architecture, featuring an interface for each layer of the conceptual model. Figure 6.2 depicts the main classes used in the implementation of the `Syhthesise` Interface. As the reader can notice the architecture presents four interfaces that are used by the `Syhthesise` Interface:

- `MWAbstractionProvider`, which represents the *Abstraction* layer in Figure 2.1.
- `OntologyMatchingProvider`, which represents the *Matching* layer in Figure 2.1.
- `MappingProvider`, which represents the *Mapping* layer in Figure 2.1, and is used in those approaches that consider matching and mapping as two separate steps, just like the approach presented in Chapter 3.
- `MatchingBasedMappingProvider`, which represents the *Matching* and the *Mapping* layer in Figure 2.1, and is used in those approaches, like the one considered in Chapter 4, which consider ontology matching and CONNECTor synthesis as parts of the same iterative process.

One of the possible behaviors of the Synthesis Enabler is reported in Figure 6.3. The `Syhthesise` Interface first invokes the performAbstraction method of `MWAbstractionProvider`, obtaining from it an instance of `MWAgnosticEnhancedLTS` for each network system. A `MWAgnosticEnhancedLTS` instance contains the information contained in an `EnhancedLTS`, augmented with the pieces of information needed to abstract the `EnhancedLTS` from its middleware implementation. After this step, the `Syhthesise` Interface invokes performMatching, getMatchingForNS1 and getMatchingForNS2 methods of `OntologyMatchingProvider`, passing as input to the first method the instances of `MWAgnosticEnhancedLTS`, obtained from the first step, and obtaining from the second and the third invoked methods an instance of `OntologyMediatedMWAgnosticEnhancedLTS` for each of the network systems. An instance of `OntologyMediatedMWAgnosticEnhancedLTS` for a network system ns1, is an instance of `MWAgnosticEnhancedLTS`, whose alphabet has been replaced by an alphabet in common with ns2, and computed using the ontology. After these two steps, the `Syhthesise` Interface can invoke the produceMapping method of `MappingProvider`, passing as input the instances of `OntologyMediatedMWAgnosticEnhancedLTS` retrieved from `OntologyMatchingProvider`, and obtaining a CONNECTor, described as an instance of `ColouredAutomaton`.

Since the implementations of the CONNECTor theory, presented in Chapters 3 and 4, present a main difference in the way they expect to interact with the ontology, the Synthesis Enabler architecture should take that difference into account. The approach in Chapter 3 expects the matching phase to be performed before the mapping phase, while the approach in Chapter 4 the matching and the mapping phases are seen as part of an iterative process that produces the CONNECTor. In order to consider this case, we need to modify the behavior of the Synthesis Enabler as shown in Figure 6.4. In this case, the `Syhthesise` Interface first uses `MWAbstractionProvider`, in order to obtain two instances of `MWAgnosticEnhancedLTS` and then invokes the produceMapping method of `MatchingBasedMappingProvider`, in order to obtain a CONNECTor.

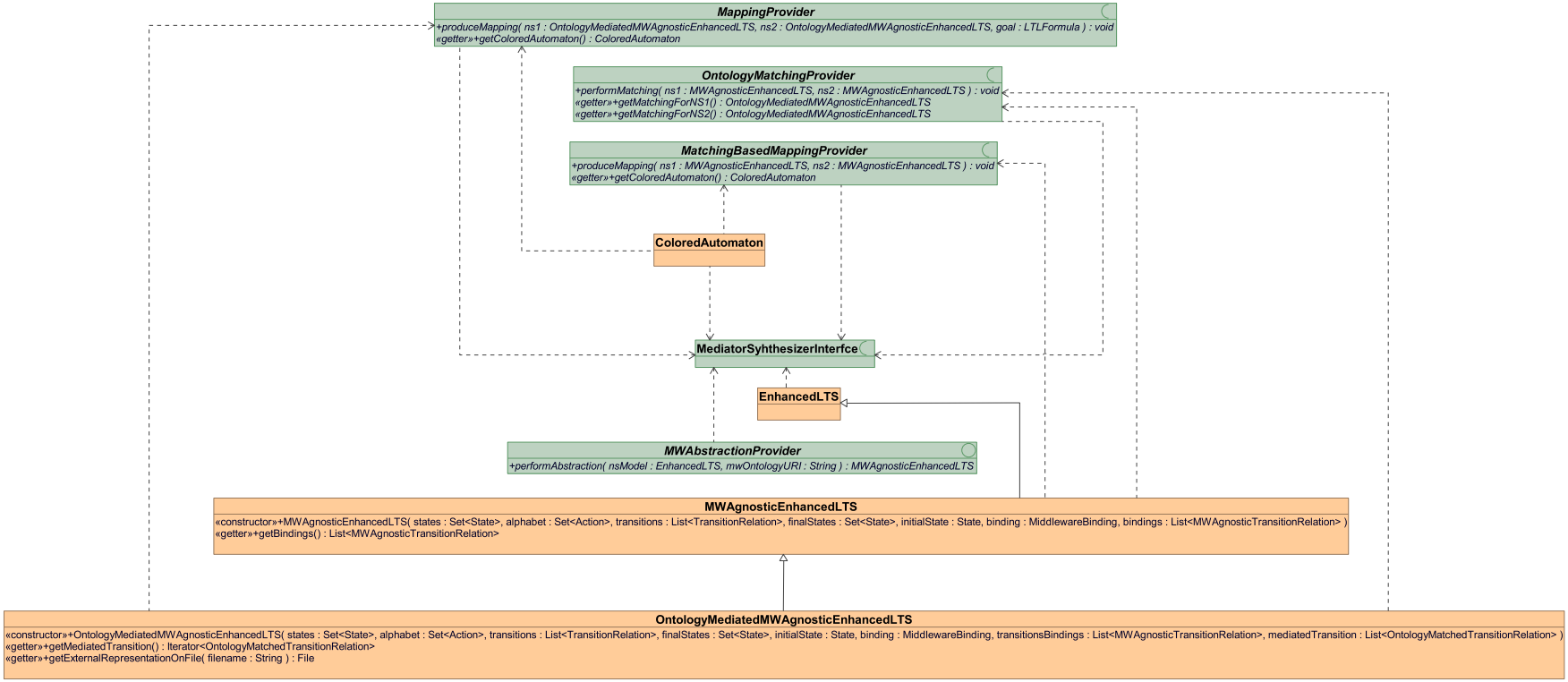

Figure 6.2: Architecture of the Synthesis Enabler.

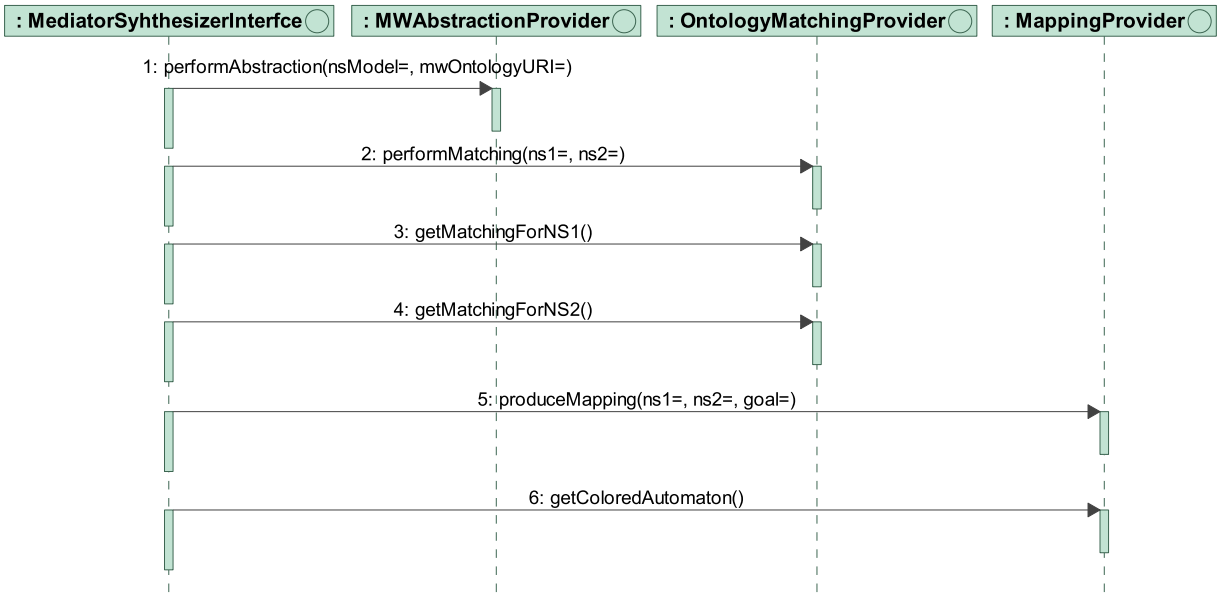

Figure 6.3: Behavior of the Synthesis Enabler, using `OntologyMatchingProvider` and `MappingProvider`.

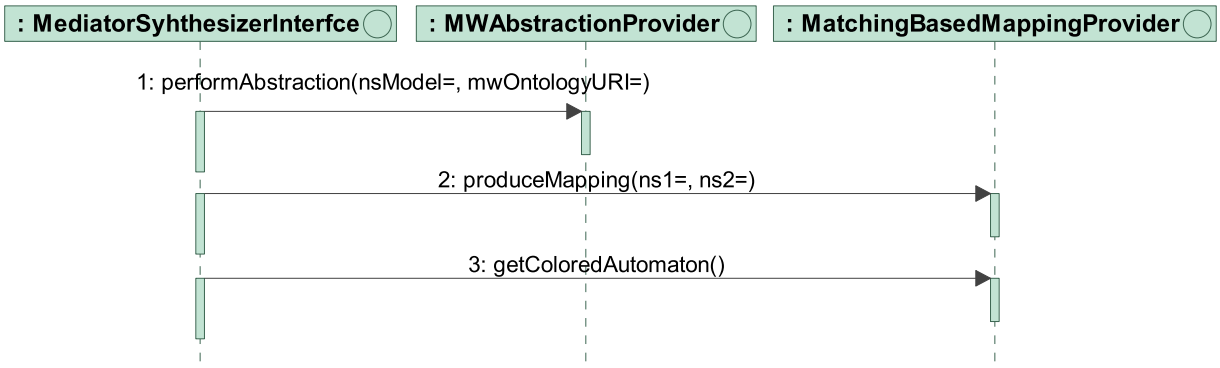

Figure 6.4: Behavior of the Synthesis Enabler, using `MatchingBasedMappingProvider` .

7 Conclusion and Future Work

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 efficient approaches for CONNECTor synthesis, which can be performed at runtime.

7.1 Contributions

In this deliverable, we have presented the results achieved during Year 3 that have allowed us to define a concrete architecture and related API, together with its actual implementation, for the Synthesis enabler.

This deliverable contributes the following achievements to CONNECT:

- As a first achievement we have revised the theory of mediators devised during the first two years of the projects by introducing the handling of actions with data and by enhancing the formal definition of protocol abstraction and matching.
- As a second major achievement of this deliverable is the implementation of the CONNECTor theory in two different approaches, which were shown to be complementary in Chapter 5: a goal-based approach and a mapping-based one. The goal-based approach considers the intent to connect two systems to drive the synthesis of a CONNECTor, as suggested by the second review recommendations, presented in Section 1.3. However, this approach produces partial models of the abstract CONNECTor. This limitation is overcome by the mapping-based approach. Moreover, both implementations model networked systems as automata with data, thus aligning with the work on connector models in WP2 and on register automata in WP4.
- The third and last major achievement of this deliverable is the implementation of a synthesis framework for the synthesis of mediators. The framework reunites the two implementations of the CONNECTor theory, and gives the possibility to select one or the other, according to the presence of an intent to connect the two networked systems.

7.2 Future Work

As future work, we intend to finalize the implementation of the Synthesis enabler in order to cope with the various possible mismatches identified by our theory of mediators. More specifically, we intend to enhance our synthesis methods and their related implementations in order to support the *extra receive*/*missing send* mismatch, which is the only one that our current implementations cannot handle. Furthermore, we have so far concentrated on the synthesis of mediators from scratch, while the construction of mediators by composing existing ones would enable more efficient synthesis and support self-adaptive emergent middleware. Ongoing CONNECT research on the compositional specification theory and connector algebra formalized within the work of WP2 [5, 6] will provide us the required foundations to enable the dynamic adaptation of a synthesized CONNECTor to changes in the CONNECTed NSs or specified goal. Finally, by referring to the work carried on WP5, we plan to enhance our CONNECTor synthesis methods to consider non-functional aspects of the NSs interaction. The approach that we have in mind, by using non-functional estimation and optimization techniques, shall produce the best connector with respect to specified non-functional requirements for the system to be connected. In case more than one non-functional aspect is considered, a trade-off analysis (based on multi-objective function optimization) needs to be executed.

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

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

[6] CONNECT consortium. CONNECT Deliverable D2.3: Rephrasing interoperability in terms of connector behaviours. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[7] 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/.

[8] 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/.

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

[10] CONNECT consortium. CONNECT Deliverable D3.3: Dynamic connector synthesis: revised prototype implementation, Appendix - Prototype. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[11] CONNECT consortium. CONNECT Deliverable D5.3: Consolidated Dependability Framework. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[12] U. Aßmann, S. Zschaler, and G. Wagner. Ontologies, Meta-models, and the Model-Driven Paradigm. Springer, 2006.

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

[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] M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi. SMT-based verification of ltl specification with integer constraints and its application to runtime checking of service substitutability. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 244 –254, 2010.

[16] M. M. Bersani, A. Frigeri, A. Morzenti, M. Pradella, M. Rossi, and P. S. Pietro. Bounded reachability for temporal logic over constraint systems. In *TIME*, 2010.

[17] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010.

[18] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In *Middleware'11*, 2011.

[19] A. Brogi, C. Canal, and E. Pimentel. Component adaptation through flexible subservicing. *Sci. Comput. Program.*, 63(1):39–56, 2006.

[20] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *In Proceedings of ICSOC*, 2006.

[21] Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Proc. ICDCS*, 2011.

[22] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Proc. Middleware*, 2009.

[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] C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 34(4):546–563, 2008.

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

[27] H. Chang, L. Mariani, and M. Pezzè. In-field healing of integration problems with cots components. In *ICSE*, pages 166–176, 2009.

[28] E. Cimpian and A. Mocan. WSMX process mediation based on choreographies. In *Proceedings of Business Process Management Workshop*, 2005.

[29] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[30] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 268–279, London, UK, 1998. Springer-Verlag.

[31] S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Model-checking CTL* over flat Presburger counter systems. *Journal of Applied Non-Classical Logics*, 20(4):313–344, 2010.

[32] 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.

[33] A. Finkel and A. Sangnier. Reversal-bounded counter machines revisited. In *Proceedings of the 33rd international symposium on Mathematical Foundations of Computer Science*, MFCS '08, pages 323–334, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ws-engineer: A model-based approach to engineering web service compositions and choreography. In *Test and Analysis of Web Services*, pages 87–119. 2007.

[35] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

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

[37] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[38] F. Giunchiglia, M. Marchese, and I. Zaihrayeu. Encoding classifications into lightweight ontologies. *J. Data Semantics*, 2007.

[39] 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.

[40] Hamid R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *In Proceedings of WWW '07*, 2007.

[41] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25:116–133, January 1978.

[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] P. Inverardi, R. Spalazzese, and M. Tivoli. Application-layer connector synthesis. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659, pages 148–190, 2011.

[44] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Journal of Systems and Software*, 65(3):173–183, 2003.

[45] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*. 2011.

[46] ITU-T. Message sequence chart (msc), geneva recommendation. http://www.itu.int/ITU-T/studygroups/com17/languages/Z120.pdf.

[47] 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.

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

[49] 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.

[50] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 1976.

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

[52] S. S. Lam. Protocol conversion. *IEEE Transaction Software Engineering*, 1988.

[53] 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.

[54] M. Bersani amd A. Frigeri and M. Pradella and M. Rossi. Zot: a bounded model/satisfiability checker. http://home.dei.polimi.it/pradella/Zot/.

[55] J. Magee and J. Kramer. *Concurrency: State models and Java programs*. Hoboken (N.J.): Wiley, 2006.

[56] J. A. Martìn and E. Pimentel. Automatic generation of adaptation contracts. In *Proceedings of FOCLASA*, 2008.

[57] F. Menge. Enterprise Service Bus. In *Proc. Free and open source software conf.*, 2007.

[58] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[59] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[60] M. Minsky. Recursive unsolvability of post's problem of 'tag'. *Annals of Mathematics*, 74:437–453, 1961.

[61] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *WWW*, 2010.

[62] N. Noy and E. Wallace. Simple part-whole relations in owl ontologies. http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/, 2005.

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

[64] M. Presburger and D. Jabcquette. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 2, issue 2:225–233, 1991.

[65] D. Preuveneers and Y. Berbers. Prime numbers considered useful: Ontology encoding for efficient subsumption testing, 2006.

[66] 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.

[67] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*, volume 35. Elsevier Science, 2006.

[68] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[69] R. Spalazzese. *A Theory of Mediating Connectors to achieve Interoperability*. PhD thesis, University of L'Aquila, April 2011.

[70] R. Spalazzese and P. Inverardi. Components interoperability through mediating connector pattern. In *WCSI 2010, arXiv:1010.2337; EPTCS 37, 2010, pp. 27-41*.

[71] R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *ECSA*, 2010.

[72] 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.

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

[74] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *ICSE*, pages 374–384, 2003.

[75] 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.

[76] M. Tivoli, P. Fradet, A. Girault, and G. Gößler. Adaptor synthesis for real-time components. In *TACAS*, pages 185–200, 2007.

[77] R. Vaculín, R. Neruda, and K. P. Sycara. The process mediation framework for semantic web services. *IJAOSE*, 2009.

[78] W3C. Semantic annotations for WSDL and XML schema. sawsdlhttp://www.w3.org/TR/sawsdl/.

[79] W3C. Simple part-whole relations in owl ontologies. http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/.

[80] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Trans. Softw. Eng.*, 24(5):331–341, 1998.

[81] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 1992.

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

[83] Z. Wu, K. Gomadam, A. Ranabahu, A. P. Sheth, and J. A. Miller. Automatic composition of semantic web services using process mediation. In *ICEIS (4)*, 2007.

[84] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 1997.