Modeling of application- and middleware-layer interaction protocols

Amel Bennaceur, Antonia Bertolino, Paul Grace, Paola Inverardi, Valérie Issarny, Massimo Tivoli

To cite this version:

Amel Bennaceur, Antonia Bertolino, Paul Grace, Paola Inverardi, Valérie Issarny, et al.. Modeling of application- and middleware-layer interaction protocols. [Technical Report] 2010. inria-00464661

HAL Id: inria-00464661

https://hal.inria.fr/inria-00464661

Submitted on 17 Mar 2010

CONNECT

Emergent Connectors for Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D3.1

Modeling of application- and middleware-layer interaction protocols

http://www.connect-forever.eu

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

Deliverable Number	:	D3.1
Title of Deliverable	:	Modeling of application- and middleware-layer interaction protocols
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.2
Contractual Delivery Date	:	1 February 2010
Actual Delivery Date	:	15 February 2010
Contributing WPs	:	WP3
Editor(s)	:	Massimo Tivoli (UNIVAQ)
Author(s)	:	Amel Bennaceur (INRIA), Antonia Bertolino (CNR), Paul Grace (LANCS), Paola Inverardi (UNIVAQ), Valérie Issarny (INRIA), Romina Spalazzese (UNIVAQ), Massimo Tivoli (UNIVAQ)
Reviewer(s)	:	Animesh Pathak (INRIA), Paul Grace (LANCS)

Abstract

The CONNECT *Integrated Project aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from the networked environment.* CONNECT *aims at dropping the interoperability barrier by adopting a revolutionary approach to the seamless networking of digital systems, that is, synthesizing on-the-fly the connectors via which networked systems communicate. The resulting emergent connectors are effectively synthesized according to the behavioral semantics of application- down to middleware-layer protocols run by the interacting parties. The role of work package WP3 is to devise automated and compositional approaches to connector synthesis, which can be performed at run-time. Given the respective interaction behavior of networked systems, we want to synthesize the behavior of the connector(s) needed for them to interact. These connectors serve as* mediators *of the networked systems' interaction at both application and middleware layers. In this deliverable, we set the scene for a formal theory of the automated synthesis of application- and middleware-layer protocol mediators. We formally characterize mediating connectors between mismatching application-layer protocols by rigorously defining the necessary conditions that must hold for protocols to be mediated. The outcome of this formalization is the definition of two relationships between heterogenous protocols:* matching *and* mapping*. The former is concerned with checking whether a mediator letting two protocols interoperate exists or not. The latter concerns the algorithm that should be executed to synthesize the required mediator. Furthermore, we analyze the different dimensions of interoperability at the middleware layer and exploit this analysis to formalize existing solutions to middleware-layer interoperability. Since the work on application-layer mediator synthesis is based on the assumption that a model of the interaction protocol for a networked system is dynamically discovered, we finally present an approach, based on data-flow analysis and testing, for the automated elicitation of application-layer protocols from software implementations. This approach presents similarities, but also several differences, with the work of work package WP4 (protocol learning). Furthermore, it allowed us to proceed in parallel with the work of WP4 and to state the requirements that the learning approaches have to satisfy to enable mediator synthesis. For this reason, we keep this work separate from the work on protocol learning and discuss it in this deliverable. All the approaches mentioned above are applied to several examples and scenarios.*

Keyword List

Connectors, Protocol Mediators, Protocol Specification, Protocol Synthesis, Protocol Elicitation, Application-Layer Interoperability, Middleware-Layer Interoperability, Data-Flow Analysis, Testing.

Document History

Version	Type of Change	Author(s)
0.1	Outline and planning	Massimo Tivoli (UNIVAQ)
0.2	First version of Chapter 5	Massimo Tivoli (UNIVAQ)
0.3	First version of Chapters 2, 3, and 4	Amel Bennaceur (INRIA), Valérie Issarny (INRIA), Romina Spalazzese (UNIVAQ), and Paola Inverardi (UNIVAQ)
0.4	Second version of Chapters 3 and 4	Amel Bennaceur (INRIA), Romina Spalazzese (UNIVAQ)
0.5	Second version of Chapters 2 and 5	Massimo Tivoli (UNIVAQ)
0.6	Revision of Chapters 2,3, and 4; editing of Abstract, Keywords, Chapter 1, and Chapter 6	Amel Bennaceur (INRIA), Paola Inverardi (UNIVAQ), Romina Spalazzese (UNIVAQ), Massimo Tivoli (UNIVAQ)
0.7	Complete revision of the content of v0.6	Amel Bennaceur (INRIA)
0.8-1.2	Internal Reviews	Amel Bennaceur (INRIA), Antonia Bertolino (CNR), Paul Grace (LANCS), Paola Inverardi (UNIVAQ), Valérie Issarny (INRIA), Animesh Pathak (INRIA), Romina Spalazzese (UNIVAQ), and Massimo Tivoli (UNIVAQ)

Table of Contents

List of Figures

List of Tables

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from the networked environment. At present, the efficacy of integrating and composing networked systems depends on the level of interoperability of the systems's underlying technologies and in particular embedded middleware. Still, middleware-based interoperability cannot cover the ever growing heterogeneity dimensions of the networked environment. CONNECT then 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 (or CONNECTors) are effectively synthesized according to the behavioral semantics of application- down to middleware-layer protocols run by the interacting parties.

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

- **Synthesis of application-layer conversation protocols**. The goal here is to identify connectors patterns that allow the definition of methodologies to automatically synthesize, in a compositional way and at run-time, application-layer CONNECTors.
- **Synthesis of middleware-layer protocols**. Our objective here is to generate adequate *protocol translators* (mappings) that enable heterogeneous middleware to interoperate, and realize the required non-functional properties, thus successfully interCONNECTing networked systems at the middleware level.
- **Model-driven synthesis tools**. In this subtask, we exploit model-to-model and model-to-code transformation techniques to automatically derive, at run-time, a CONNECTor's actual code from its synthesized model. This step should guarantee the *correctness-by-construction* of the CONNECTors' implementations with respect to the functional and non-functional requirements of the networked applications that are made interoperable through the CONNECTors.

Emergent connectors, mentioned above, act as *mediators* for today's and future systems that increasingly need to be connected. The mediator concept has been introduced to deal with different heterogeneity dimensions spanning: (i) terminology (data level mediation), (ii) representation format and transfer protocols (combination of data level and protocol mediations), (iii) functionality (behavioral type mediation) and (iv) application-layer protocols (mediation of behavioral mismatches occurring during interactions) [72]. A key challenge for today's system architectures is to embed the necessary support for automated mediation, i.e., the connector concept needs to evolve towards the one of *mediating connector*. Automated mediation has deserved a great deal of attention in all the aforementioned heterogeneity dimensions. Considering today's state of the art, ontologies appear as the core concept to deal with data heterogeneity, logic-based formalisms stand as the natural paradigm for overcoming functional heterogeneity, and process algebras are obvious candidates for reasoning about protocol mediation. Still, enabling reasoning and further solving of semantic mismatches at run-time, while not over-constraining the ability to communicate, remain open research questions.

The work described in this deliverable represents a first step towards the achievement of the above objectives. In this deliverable, we more specifically concentrate on the issue of enabling automated protocol mediation. In that context, we will interchangeably use the terms mediating connector, mediator and CONNECTor in the following. Our work over the reporting period has in particular lead to the definition and formalization of complex protocol matching and mapping relationships over application-layer protocols. This contribution is described in Chapter 3 and is illustrated at work on two examples: one concerns interoperability between two heterogeneous instant messaging systems, while the other concerns the application of the approach to the *Popcorn scenario* provided by WP1 and described in Deliverable

D1.1 [2]. The defined relationships represent two essential operations for the dynamic synthesis of mediating connectors to enable eternal networked systems. In fact, the matching relationship allows the rigorous characterization of the conditions that must hold in order for two heterogeneous protocols to be able to interoperate through a mediator. Thus it allows one to state/check the existence of a mediator for two heterogeneous protocols. The mapping relationship introduces the formal specification of the algorithm that should be performed in order to automatically synthesize the required mediator.

Concerning middleware-layer protocols, in Chapter 4, we first analyze the different dimensions of middleware-layer interoperability. This analysis allows us to formalize existing solutions to middleware-layer interoperability, presented in [2] and assess the one based on dynamic protocol synthesis as aimed by CONNECT through two examples taken from the *Popcorn scenario*. This leads us to conclude that existing solutions to the dynamic synthesis of interoperable middleware protocols do not address overall CONNECT requirements, especially missing interoperability among middleware of different types. Then, we evaluate the applicability of the aforementioned approach to application-layer interoperability at the middleware-layer concluding that the approach devised so far still needs some adjustments to be effective for both application- and middleware-layer interoperability.

Furthermore, since the work on application-layer mediator synthesis is based on the assumption that a model of the interaction protocol for a networked system is dynamically discovered, we finally present a model-driven approach, based on data-flow analysis and testing, to the automated elicitation of application-layer protocols from software implementations. So far, the defined approach has been applied to the context of Web services. This approach is described in Chapter 5 and is applied to an existing Web service, which is the *Amazon E-Commerce Service*. Together with the above mentioned protocol matching and mapping relationships, this approach represents another step towards the dynamic synthesis of mediating connectors. This approach presents similarities, but also several differences, with the work of work package WP4 (protocol learning). Furthermore, it allowed us to proceed in parallel with the work of WP4 and to state the requirements that the learning approaches have to satisfy to enable mediator synthesis. For this reason, we keep this work separate from the work on protocol learning and discuss it in this deliverable.

As detailed in the rest of this deliverable, the progress made with respect to WP3's objectives reported above can be summarized as follows:

- Formalization of matching and mapping relationships for application-layer interaction protocols, and of the corresponding CONNECTor generation algorithm.
- Identification of the application-layer protocol mismatches that can occur/be solved.
- Characterization of the different dimensions of middleware-layer interoperability.
- Formalization of existing solutions to middleware-layer interoperability.
- Characterization of the *pros* and *cons* concerning the applicability, at the middleware-layer, of the mediator synthesis for application-layer interoperability.
- Algorithm for the automated elicitation of application-layer protocols.

This deliverable is organized as follows. Chapter 2 sketches the background for our work, studying the paradigm of protocol mediation and further analyzing formal foundations to reason upon such mediation. In Chapter 3, we formalize the theory underlying the automated mediation of *application-layer* protocols, and apply it to two scenarios: *instant messaging protocols* and the *Popcorn scenario*. Mediator synthesis for middleware-layer protocols is discussed in Chapter 4. Chapter 5 presents an automated approach to protocol elicitation and applies it to an existing Web service. Chapter 6 concludes and discusses future work.

2 Foundations for CONNECTor Synthesis

In Sections 2.1 we discuss the mediation paradigm and the concept of *mediating connector* providing background notions for it and for the process of automated mediation. In Section 2.2, we introduce the relevant formal foundations for connector synthesis. Finally, in Section 2.3, we conceptually characterize two aspects that are crucial for enabling effective automated mediator synthesis, i.e., the matching and mapping of functionalities of heterogeneous networked systems.

All these notions serve as background for the full understanding of the theory of mediators introduced in Chapter 3, for the middleware-layer protocol formalization presented in Chapter 4, and for the protocol elicitation method described in Chapter 5.

2.1 From Mediation to CONNECTors

The mediation paradigm, underlying the definition of CONNECTors and their automated synthesis, encompasses a number of architectural paradigms like adapter, bridge and wrapper. This section provides a brief definition of the concept as used in this document. We then survey protocol mediation patterns that have been elicited in the literature together with approaches to automated mediation.

The mediator concept was initially introduced to cope with the integration of heterogeneous data sources [85, 84] and as a design pattern [28]. However, with the significant development of Web technologies and given the ability to communicate openly for networked systems, many heterogeneity dimensions arise and need be mediated [26]:

- Mediation of data structures allows for data to be exchanged according to semantic matching, as opposed to requiring syntactic matching and further usage of identical data formats;
- Mediation of functionalities enables one to discover the location of networked resources that provide a required functionality (in isolation and/or in combination) based on semantic matching and possible adaptation;
- Mediation of business logics enables networked resources that provide complementary functionalities to be connected together although they may execute interaction protocols whose respective behaviors do not match;
- Mediation of message exchange protocols supports the actual interaction among networked resources although they may use different middleware protocols for communication. Middleware heterogeneity ranges from heterogeneity of implementations to that of distributed computing models and related coordination models and extra-functional properties.

Facing this heterogeneity, mediation architectures embed a number of enablers [72]:

- Data level mediation primarily relies on techniques for ontology integration [59], dealing with the mapping, alignment, and merging of ontologies;
- Functional mediation may be based on logical relationships between functional descriptions of networked resources that are expressed in terms of pre- and post-conditions over the resources' states [71];
- Business logic and protocol mediation is concerned with the mediation of protocols from the application (possibly) down to the middleware layers. It strives to build techniques to solve behavioral mismatches among protocols run by interacting parties. As discussed below, proposed solutions introduce algorithms that establish a valid process for interaction given the respective processes run by the interacting parties. The challenge is then to promote flexibility by dynamically solving behavioral mismatches as far as the connected resources functionally match.

Automated mediation has deserved attention in all the aforementioned heterogeneity dimensions. This especially holds in the context of Web services technologies that is certainly one of today's most popular and enabling architectures for networked resources. Still, enabling reasoning and further solving the semantic mismatches at runtime, while not over-constraining the ability to communicate, remains an open research question. Focusing on automated protocol mediation, solutions rely on:

- The adequate modeling of processes abstracting the behavior of the protocols to be bridged, where finite state machines is the modeling formalism of choice in most work in light of their flexibility and applicability;

- The definition of a matching relationship between the process models that sets the conditions under which protocol interoperability is supported; and

- The elicitation of an algorithm that computes an appropriate mapping between matching process models.

A base approach towards automated protocol mediation is to categorize the various types of protocol mismatches that may occur and that must be solved, according to the structure of the associated processes, and then to define corresponding mediation patterns. Five basic patterns have been introduced in the literature in the context of Web services [23, 12]. These concern: (i) stopping an unexpected message, (ii) inverting the order of messages, (iii) splitting a message, (iv) combining messages, and (v) sending a dummy acknowledgment. Given the mediation patterns, custom mediators may be designed through the assembly of relevant patterns according to the behavioral mismatches identified among the protocols to be made interoperable. Such an issue is in particular addressed in [42], which provides tools to developers to assist them to identify protocol mismatches and to compose mediators. However, this remains quite limited with respect to enabling interoperability in today's networking environments that are highly dynamic. Indeed, mediators need to be synthesized on-the-fly so as to allow interactions with networked systems that are not known in advance. Such a concern is in particular recognized by the Web service research community that has been studying solutions to the automated mediation of business processes in the recent years.

A number of solutions to automated protocol mediation have recently emerged, leveraging the rich capabilities of Web services and Semantic Web technologies [80, 79, 55, 86]. They differ with respect to:

- A priori exposure of the process models associated with the protocols that are executed by networked resources, thus possibly requiring to learn model on-the-fly, if not part of the networked systems' interfaces;

- A priori knowledge about the protocols run by the interacting parties, thus possibly enabling to synthesize part of the mediator off-line; and

- The matching relationship that is enforced, possibly weakening flexibility to alleviate the complexity of mediation.

However, most solutions are discussed informally, making it difficult to assess their respective advantages and drawbacks. They further remain rather vague on the definition of the enforced matching relationship. Hence, what is needed is a formal foundation for *mediating connectors* from which:

- Protocol matching and associated mapping relationships may be rigorously defined and assessed; and

- The above relationships may be automatically reasoned upon, thus paving the way for on-the-fly synthesis of mediating connectors.

In that direction, [87] proposes a theory to characterize and solve the interoperability problem of augmented interfaces of applications. The authors formally define the checks of applications compatibility and the concept of adaptors. The latter can be used to bridge the differences discovered while checking the applications that have functional matching but are protocol incompatible. Furthermore they provide a theory for the automated generation of adaptors based on interface mapping constraints. One main disadvantage of this work is that the approach is semi-automated because the interface mapping must be specified manually by the architect of the adaptor. Additionally, applications are assumed to agree on the ordering of messages, thus not solving ordering mismatches.

A recent work [24] addresses the interoperability problem between services and provides experimentation on real Web 2.0 social applications. The paper deals with the integration of a new service instance with the same functionality as the instance it substitutes, but having a different implementation that does not still guarantee behavioral compatibility despite complying with the same API of the previous one. They hence propose a technique to dynamically detect and fix interoperability problems based on a catalogue of inconsistencies and their respective adaptors. Still, the approach is not fully automated since although the mismatches are discovered and the corresponding adaptors are selected dynamically, the identification of mismatches and of the opportune adaptors is made by the engineer.

Our work then contributes to the issue of automated protocol mediation by targeting the fully automated synthesis of mediating connectors. Towards that objective, we introduce a formal theory of CONNECTors that enables reasoning upon protocol matching and mapping.

2.2 Formal Foundations for CONNECTors

One of the issues and challenges of CONNECT is to elicit an adequate modeling of the processes abstracting both the behavior of the protocols to be bridged and of CONNECTors. Moreover, such models have to underpin the automated reasoning about the interacting protocols run by networked systems to enable them to interoperate, and CONNECTor behaviors. Towards this direction, in the following, we analyze the formal foundations of interaction protocol specification.

In the context of CONNECT, we call "*interaction protocol*" the behavior of a networked system in terms of the messages it exchanges with its environment (i.e., the other networked systems). Concerning the specification of interaction protocols, a natural way of describing them is using *Labelled Transition Systems* (LTSs) [39]. LTSs constitute a fundamental model of concurrent computation that is widely used in light of its flexibility and applicability. LTSs are often used as a semantic model for many expressive and formal behavioral languages such as *process algebras* (also called process calculus).

Process algebras are used to model concurrent systems. Example of these languages are CCS [51], CSP [63], FSP [45] π-Calculus [52, 53], and Kell Calculus [15, 16, 65, 70], just to mention a few of them. Process algebras provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes (e.g., using *bisimulation* [51]). As mentioned above, often these calculi are formalized operationally by using an LTS-based semantics [81]. However, process algebras can be used to define interaction protocols in a more concise way.

To explain the correspondence, as a semantic model, between LTSs and process algebras, let us briefly recall the key characteristics of a specific process calculi, i.e., CCS (*Calculus of Communicating Systems*). We refer to [51] for more details. It is worthwhile noticing that the semantic correspondence that we explain in the following can be analogously defined between LTSs and another process algebra different from CCS, e.g., FSP.

The CCS syntax is the following:

$$p ::= nil \mid \mu.p \mid p + p \mid p|p \mid p\backslash A \mid x \mid p[f]$$

Terms ($Terms$) generated by p are called *process terms* (or simply *processes*, or *terms*); x ranges over a set $\{X, Y, \ldots\}$, of process variables. A process variable is defined by a process definition $x \overset{def}{=} p$, (p is called the expansion of x). As usual, there is a finite set of visible actions $Vis = \{a, \overline{a}, b, \overline{b}, \ldots\}$ over which α ranges, while μ, ν range over $Act = Vis \cup \{\tau\}$, where τ denotes the so-called *internal action*. We denote by $\overline{\alpha}$ the action complement: if $\alpha = a$, then $\overline{\alpha} = \overline{a}$, while if $\alpha = \overline{a}$, then $\overline{\alpha} = a$. By nil we denote the empty process. The operators to build process terms are prefixing ($\mu.p$), summation ($p + p$), parallel composition ($p|p$), restriction ($p\backslash A$) and relabelling ($p[f]$), where $A \subseteq Vis$ and $f : Vis \rightarrow Vis$.

An operational semantics OP is a set of inference rules defining a relation $D \subseteq Terms \times Act \times Terms$. The relation is the least relation satisfying the rules. If $(p, \mu, q) \in D$, we write $p \xrightarrow{\mu}_{OP} q$. The rules defining the semantics of CCS [51], from now on referred to as *Structural Operational Semantics* (SOS), are here recalled:

$$\begin{array}{llll}
Act & \dfrac{}{\alpha.P\overset{\alpha}{\rightarrow}P} & Synch & \dfrac{P\overset{\alpha}{\rightarrow}P',Q\overset{\overline{\alpha}}{\rightarrow}Q'}{P|Q\overset{\tau}{\rightarrow}P'|Q'} \\[2ex]
Sum & \dfrac{P\overset{\alpha}{\rightarrow}P'}{P+Q\overset{\alpha}{\rightarrow}P'} & Rel & \dfrac{P\overset{\alpha}{\rightarrow}P'}{P[f]\overset{f(\alpha)}{\rightarrow}P'[f]} \\[2ex]
Comp & \dfrac{P\overset{\alpha}{\rightarrow}P'}{P|Q\overset{\alpha}{\rightarrow}P'|Q} & Res & \dfrac{P\overset{\alpha}{\rightarrow}P',\alpha\notin L\cup\overline{L}}{P\backslash L\overset{\alpha}{\rightarrow}P'\backslash L} \\[2ex]
Con & \dfrac{P\overset{\alpha}{\rightarrow}P',A\overset{def}{=}P}{A\overset{\alpha}{\rightarrow}P'} & &
\end{array}$$

The rules Sum and $Comp$ have a symmetric version which is omitted.

An LTS L is a quadruple (S, T, D, s_0), where S is a set of states, T is a set of transition labels, $s_0 \in S$ is the initial state, and $D \subseteq S \times T \times S$ is a transition relation. A transition system is finite if D is finite.

A finite computation of a transition system is a sequence $\mu_1\mu_2 \dots \mu_n$ of labels such that: $s_0 \xrightarrow{\mu_1}_{OP} \dots \xrightarrow{\mu_n}_{OP} s_n$.

Given a term p (and a set of process variable definitions), and an operational semantics OP, $OP(p)$ is the transition system $(Terms, Act, D, p)$, where D is the relation defined by OP. For example, $SOS(p)$ is the transition system defined by the SOS semantics for the term p. CCS can be used to define a wide class of systems that ranges from Turing machines to finite systems [74]; therefore, in general, CCS terms cannot be represented as finite state systems. For our purposes, in the following, we will assume that all the systems we deal with are finite state. Note that for CONNECT, this is not a restriction. We are dealing with networked systems that support a finite number of operations (e.g., (i) for a Web service relying on SOAP, its WSDL interface is defined in terms of a finite number of WSDL operations; (ii) for a COM/DCOM component, its IDL interface is defined in terms of a finite number of IDL methods). In our model, each operation of a networked system can be seen as a point of interaction of the system with its expected environment (e.g., an observable action of an automaton). If we would model all the possible externally observable system interactions with an automaton, what matters about a particular interaction is not whether it drives the automaton into an accepting state (since we cannot detect this due to the black-box nature of the system) but whether the automaton is able to perform the corresponding sequence of actions interactively. Thus, we should consider an automaton in which every state is an accepting state [51, 33], i.e., an LTS. A consequence is that if an automaton accepts a particular interaction seen as a sequence of system operation invocations, then it also accepts any initial part of that interaction. In other words, due to the finiteness of the set of system operations, although all the possible system interactions can be infinite, we can always finitely represent them since the language built over the system operations (i.e., the model of the system's interaction protocol) is prefix-closed [33]. Prefix-closed languages are generated by prefix-grammars that describe exactly all regular languages. It is well-known that regular languages are always accepted by finite-state automata. Thus, for us, it is sufficient to consider finite state systems for dealing with all the systems we are interested in.

If we assume to deal with finite state systems, then a correspondence between CCS terms and LTSs can be always defined. A CCS term may be encoded in LTS as follows:

- LTS states are CCS terms;
- transitions are given by $\longrightarrow_{OP}$, i. e. by operational semantics;
- the LTS start state is the one corresponding to the encoded CCS term;

and any finite-state LTS can be encoded in CCS as follows:

- associate a process S_i to each LTS state s_i;
- in the declaration of S_i, sum (summation operator $+$) together terms of form $\alpha.S_j$ for each transition $s_i \overset{\alpha}{\rightarrow} s_j$ in LTS;
- the CCS term is the one corresponding to the encoded LTS start state.

The work discussed in Chapter 3 considers LTSs as the formal tool for modeling application-layer protocols. Although from the previous discussion it is clear that we could equivalently choose a process algebra, e.g., CCS or FSP, we prefer using LTSs since, in any case, the CONNECTor synthesis algorithm deals with data structures that encode LTSs.

On the other hand, specifications for middleware-layer protocols are often described using process algebra, and in particular FSP [69]. Thus, the work described in Chapter 4 considers also FSP as the formal tool for formalizing middleware-layer protocols. Furthermore, for the purposes of the work described in Chapter 4, there is the need to check the correctness of the provided formalization. Using FSP allows us to exploit the LTSA tool [45] in order to automatically perform this correctness check.

However, note that the semantics of FSP can be expressed using LTSs [45] in a way analogous to what has been discussed above. Thus note that, despite the different process algebra notations that one could choose depending on, e.g., his own expertise and the purposes of the work, the underlying semantic model can be always expressed by using the same formal tool, i.e., LTSs.

FSP syntax	**Description**
$a{\rightarrow}P$	action prefix
$a{\rightarrow}P \mid b{\rightarrow}Q$	choice
$P \| Q$	parallel composition
$label{:}P$	process labelling
$P/\{new/old\}$	relabelling
$P\backslash\{hidden\}$	hiding
$when(n{<}T)\ a{\rightarrow}P$	guarded action
$P{+}\{a{,}b{,}c\}$	alphabet extesion
$STOP{,}ERROR$	predefined processes
$set\ S{=}\{a{,}b{,}c\}$	defines a set S
$range\ R{=}0..5$	defines a range R
$[v{:}S]$	binds variable v to a value chosen from S

Table 2.1: An overview of FSP operators

A quick reference for some FSP operators is shown in Table 2.1, for further information see [45]. Processes describe actions (events) that occur in sequence, and choices between event sequences. Each process has an alphabet of the events that it is aware of (and either engages in or refuses to engage in). When composed in parallel, processes synchronize on shared events: if processes P and Q are composed in parallel as $P||Q$, events that are in the alphabet of only one of the two processes can occur independently of the other process, but an event that is in both processes' alphabets cannot occur until both processes are willing to engage in it.

All the other process algebras that we have mentioned above present some differences and similarities with respect to CCS or FSP. For instance, the work on π-Calculus began with the need of enhancing CCS in order to achieve an algebraic formulation of the different forms of process mobility (e.g., logical and physical mobility) in distributed systems. The main idea consisted in adding a new syntactical construct, the channel, and new semantic reduction rules for the handling of channels. This led to a first version of the π-Calculus. Later, this initial version has been extended by following a high-order approach. That is, mobility can also be achieved by the powerful means of transmitting processes (and not only channels) as messages.

The Kell calculus has further been introduced, as an extension of the π-Calculus, to study programming models for wide area distributed systems. It is a family of process calculi intended as a basis for studying distributed and ubiquitous component-based programming. Its aim is to support the modeling of different forms of process mobility (e.g., logical and physical mobility). This is done by considering, as it is in the π-calculus, the possibility to directly transmit processes as messages (and not only channels) plus the possibility to directly transmit *cells* that represent process locations (e.g., the IP address of a computer machine, the address of a sub-network, the ID of a local process, etc.).

2.3 CONNECT Matching and Mapping Concepts

Before embarking on the formal definition of the matching and mapping relationships among interaction protocols, which enable synthesizing CONNECTors, we explain them conceptually by referring to a particular case of matching and mapping relationships exploited by state-of-the-art work, from UNIVAQ, described in [77]. In particular, we will show that in CONNECT the concepts of matching and mapping has to go beyond the specific matching and mapping concepts the work described in [77] relies on.

A part of the problem treated in [77] can be phrased as follows: *given a set of interacting software components C, if possible, automatically derive a deadlock-free assembly A of these components*. The assembly A is realized by automatically synthesizing an additional component that is a software *coordinator* and by letting the components in C communicate only through this coordinator. The coordinator can be seen as an application-layer connector that preempts all the component interactions in order to not perform the "execution traces" always leading to deadlocks, hence restricting the set of all possible composed system's behaviors to only the behaviors that are deadlock-free. The interaction protocol performed by each of the components in C is modeled as a CCS process. As explained in Section 2.2, this means that for each component in C there is an LTS modeling the observable (from outside) behavior of the component when it interacts with its environment (i.e., all the other components of C in parallel), that is its interaction protocol.

The work described in [77] is applicable to layered software architectures (e.g., three-tier architectures). The synthesis method can be applied layer-by-layer hence reducing, without loss of generality, the application of the method to client-server architectures. One of the assumptions made by the work is that the client and the server components are already able to directly interact, although letting them interact in an uncontrolled way (i.e., without preempting their interactions by means of a suitable application-layer connector), can lead the system to deadlock. In other words they have to share at least one "complex" interaction. More precisely, at the level of their interaction protocol models, this means that the *synchronous product* [39] of their LTSs is not empty (i.e., the parallel composition of their CCS processes is different from the *nil* process).

Thus, in [77], there is the assumption that some of the functionalities of a client (resp., server) already match some of the functionalities of a server (resp., client). The matching is defined by the synchronous product of their LTSs. Thus a client (resp., server) is assumed to perform, among all the possible interactions, at least one complementary interaction with respect to an interaction of a server (resp., client). An interaction is seen as a sequence, in the component LTS, of input/output actions. Two interactions are complementary when they are the same sequence of actions and, for all the actions, the input/output type of an action in a sequence is the *complement* [51] of the input/output type of the corresponding action in the other sequence. An action corresponds to another action if they have the same label, thus the matching relationship, relatively to single actions, is simply a syntactical match between action labels.

Summing up, for the work described in [77], a simple matching relationship is implicitly defined and it is assumed to be already satisfied by the components given as input to the synthesis method. The relation is simple since, for single actions, it is just a syntactical matching relationship and, for sequences of actions, it is based on the existence of a non-empty synchronous product for those sequences. The matching relationship ensures the existence of a connector. Due to the characteristics of the considered matching relationship, the connector is synthesized in order to have a strictly sequential input-output behavior. That is, it simply routes messages (sent or received by other components) and each input action it receives is strictly followed by a corresponding output action. In other words, the matching relationship considered induces a simple input-output, one-to-one, and syntactical mapping relationship realized as an application-layer software connector.

These aspects highlight some differences between the simpler scenario assumed by the work described in [77] and CONNECT scenarios. First of all, CONNECT has to deal with both client/server and peer-to-peer architectures. Second, in CONNECT, it would be unreasonable to assume that the functionalities of the networked systems to be connected would directly match. Thus the matching relationship cannot be assumed already satisfied and, hence, it has to be efficiently checked. Third, in CONNECT the concept of functionality should not be limited to only sequences of actions in an LTS. It should be related to "complex portions" of a networked system's LTS. Furthermore, by exploiting ontological information, the action matching should not be simply defined as a one-to-one syntactical mapping but it should be a many-to-many ontological mapping. Thus, for CONNECT, more accurate (and less trivial) notions of matching and mapping need to be defined.

CONNECT Matching Relationship: defines necessary conditions that must hold in order for a set of networked systems to interoperate through a mediating connector (or CONNECTor). In our case, till now, the set is made by two networked systems and the matching condition is that they have complementary behavior. We have the assumption that the remaining functionalities are exchanged with third parties (as future work we will further investigate how to deal with third party message exchange). Moreover, two functionalities are complementary if they can be abstracted by the same model under a suitable notion of behavioral equivalence that is driven by ontological information.

Analogously to what is done in [77], if the functionalities of two networked systems match and, hence, the two networked systems perform complementary functionalities, then they can interoperate via a suitable mediating connector. In other words, if the matching relationship is satisfied, then there exists a mediating connector making the two networked systems interoperate. This mediating connector is abstracted by a mapping relationship.

CONNECT Mapping Relationship: having checked the CONNECT matching relationship, i.e., that the matching conditions hold, the mapping relationship lies in an algorithm that equates/pairs/links/maps those complementary functionalities that have behavioral discrepancies.

The formal definition of CONNECT matching and mapping relationships over interaction protocols is introduced in the next chapter (Chapter 3).

2.4 Summary

Summarizing the chapter, the concepts underlying CONNECTor synthesis are the following:

- The CONNECT project aims at solving interoperability issues concerning interaction protocols heterogeneity;
- LTSs can be considered as the formal foundation for modeling, and reasoning about, application- and middleware-layer protocols in light of its flexibility and applicability;
- The classical connector concept needs to evolve towards the one of *mediating connector* that not only coordinates the interaction behaviors of CONNECTed systems but also mediates those behaviors to enable actual interactions;
- Enabling automated mediation needs to rigorously define *matching* and *mapping* relationships between interaction protocols. The former establishes the conditions that must hold in order to state the existence of a mediating connector between heterogeneous protocols. The latter dictates how to synthesize the connector.

3 Application-layer CONNECTor Synthesis: Towards a Supporting Theory of Mediators

As discussed in the previous chapter, the automated mediation among heterogeneous protocols basically relies on: (i) the adequate modeling of the processes abstracting the behavior of the protocols to be bridged, (ii) the definition of a matching relationship between the process models that sets the conditions under which protocol interoperability is supported, and (iii) the elicitation of an algorithm that computes an appropriate mapping between matching process models synthesizing the mediator.

In this chapter, we concentrate on the automated synthesis of mediators. The mediators that we want to synthesize conform to the CONNECTor model discussed in Section 4.2 of Deliverable D1.1 [2]. We focus on the theory underlying the automated mediation of *application-layer* protocols, while mediation among *middleware-layer* protocols is discussed in Chapter 4. Thus, in this chapter, when we write “protocol” we mean “application-layer protocol”.

3.1 The Instant Messaging Example

To illustrate protocol mediation as studied within CONNECT, to give an example of what kind of protocols we deal with, and to make the theory more concrete, paving the way for automated CONNECTor synthesis, we consider the simple yet challenging example of two instant messaging systems [67].

Various instant messaging systems are now in use, facilitating communications among people. However, although those systems implement similar functionalities, end-users need to use the very same system to communicate due to behavioral mismatches of the respective protocols.

In more detail, consider Windows Messenger (WM), now called Windows Live Messenger [5], and Jabber Messenger (JM) [4]. Figure 3.1 models the respective behaviors of the associated protocols using LTSs (see Section 2.2). We use the usual convention that overlined actions denote output actions while non-overlined ones denote input actions. It is apparent that these systems should be able to interoperate since they both amount to supporting authentication of peers with their servers and then message exchanges among peers. However mediating their respective protocols to achieve interoperability is far from trivial, especially if one wants to achieve this automatically. An effort has been done in [42] to mediate instant messaging protocol mismatches allowing communication between any two clients. Unfortunately, the proposed solution requires the implementation of the translation from any client protocol (to be supported) to a reference exchange protocol to be given, and vice versa. This obviously affects the generality of the approach.

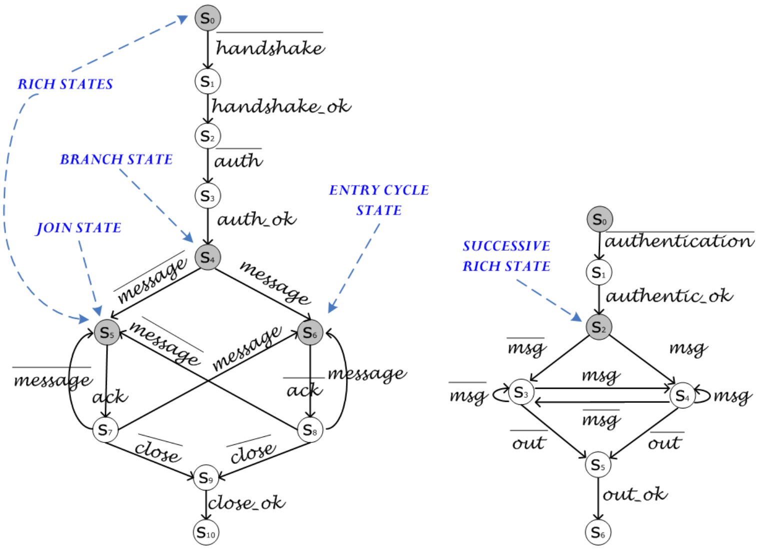
A base activity for protocol mediation is to categorize the various types of protocol mismatches that may occur and that must be solved, according to the structure of the associated processes, and then to introduce corresponding mediation patterns. As already discussed in Section 2.1, five basic patterns exist in the context of Web services that allows the resolution of several behavioral mismatches. However, the use of these patterns and their combination to achieve interoperability remains quite limited with respect to enabling interoperability in today’s networking environments that are highly dynamic.

This chapter contributes to the above by introducing the supporting formal foundation for the automated reasoning about application-layer protocol matching and mapping.

3.2 A Formalization of Protocols for Ubiquitous Connection

Starting from two protocols, we want to check if their functionalities match, i.e., if the interacting parties may coordinate and achieve their respective goals. If this is the case, then we synthesize a mediator, otherwise they cannot communicate, at least based on our methodology. The first results of our approach are in [68], which are revised and extended in this chapter.

Figure 3.2 depicts the overall idea. The basic ingredients are: (i) the behavior of two protocols represented by *LTS*s P and Q, (ii) two *ontologies* O_P and O_Q describing the meaning of P and Q actions, and (iii) a *mapping* O_{PQ} between the two ontologies. Note that when referring to protocol behavior, we mean the actions of a networked system that are observable at the interface level, i.e., its input/output actions.



(a) Windows Messenger protocol (b) Jabber Messenger protocol

Figure 3.1: LTS-based behavioral model of WM and JM protocols

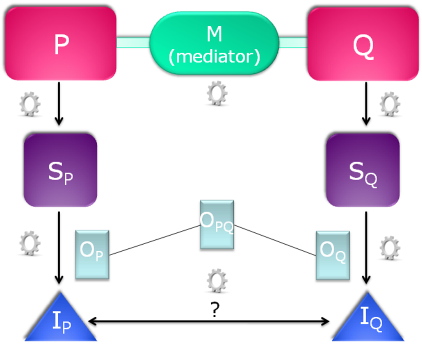


Figure 3.2: An overview of the CONNECT approach to automated mediator synthesis

We further consider protocols P and Q that are minimal where we recall that every finite LTS has a unique minimal representative; for the details of the kind of minimization that we use, the interested reader may refer to [34]. Based on the structural characteristics of the two protocols, we build an abstraction for each of them, which we call *structure*; for P and Q, it is identified in the figure by S_P and S_Q respectively. Then, using the ontology mapping function, we find the *common language* for the two protocols (pairs of words with the same meaning). This leads us to highlight the *induced LTSs* for both protocols (see I_P and I_Q), i.e., the structures where only the words belonging to the common language are highlighted. Finally, we check if the induced LTSs have a *functional matching* relation. In other words, we check if part of the provided/required functionalities of the two protocols are similar, i.e., are equivalent according to the functional matching relation we define in the following. If this is the case, then we synthesize a mediator, otherwise we cannot provide a mediator to let them communicate.

Given two protocols P and Q and a context C, the mediator M that we synthesize is such that when building the parallel composition $P||M||Q||C$, the protocols P, Q are able to communicate to evolve to their final states. This is achievable by checking that the *observable* behavior of P, Q is equivalent through a suitable notion of bisimilarity. The following details our formalization of interaction protocols and related matching.

3.2.1 Protocols as LTS

We use LTSs [39] to formally describe interaction protocols and mediators. Let Act be the universal set of observable actions (input/output actions). We get the following definition for LTS:

Definition 1 (LTS) *A LTS P is a quadruple* (S, L, D, s_0) *where:*
- S *is a finite set of states;*
- $L \subseteq Act \bigcup \{\tau\}$ *is a finite set of labels (that denote observable actions) called the alphabet of P.* τ *is the silent action. Overlined labels in* L *denote output actions while non-overlined ones denote input actions. We also use the usual convention that for all* $l \in L, l = \bar{\bar{l}}$*.*
- $D \subseteq S \times L \times S$ *is a transition relation;*
- $s_0 \in S$ *is the initial state.*

We then denote with $\{L \bigcup \{\tau\}\}^*$ the set containing all words on the alphabet L. We also make use of the usual following notation to denote transitions:

$$s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$$

We consider an extended version of LTS, which highlights the set F of the LTS' final states. For simplicity of presentation we consider that final states are states with no outgoing transitions although it is possible that final states have transitions cycling infinitely often.

An **extended LTS** is a quintuple (S, L, D, F, s_0) where the quadruple (S, L, D, s_0) is an LTS and $F \subseteq S$ and $F = \{s_f \in S : \nexists\, s_f \xrightarrow{l} s_i \in D\}$.

From now on we use the terms LTS and extended LTS interchangeably to denote the latter one.

The next concept that we need to describe is that of trace. Informally a trace is a sequence of actions of a given LTS.

Definition 2 (Trace) *Let* $P = (S, L, D, F, s_0)$*. A trace* $t = l_1, l_2, \ldots, l_m \in L^*$ *such that* $\exists (s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \ \ldots \ s_m \xrightarrow{l_m} s_n)$ *where* $\{s_1, s_2, s_3, \ldots, s_m, s_n\} \in S \wedge \forall\, 1 \leq i \leq m : (s_i, l_i, s_{i+1}) \in D$*.*
We also use the usual compact notation $s_1 \overset{t}{\Rightarrow} s_n$ *to denote a trace, where* s_1*,* s_n*, and* t *are starting state, target state, and concatenation of actions of the trace, respectively.*

The next definition illustrates the parallel composition between the protocols. Since there is a one-to-one mapping between a process P and its LTS, we use the term process and LTS interchangeably.

Given an LTS $P = (S, L, D, F, s_0)$ and $s \in S$, we identify the *configuration* $C = (S, L, D, F, s)$, in which the LTS is in s. Given $s' \in S$ and $a \in L$, we say that P *changes configuration* by transiting with the action a from a configuration C into another configuration C' if $(s, a, s') \in D$. Formally: $(S, L, D, F, s) \xrightarrow{a} (S, L, D, F, s') \ if \ (s, a, s') \in D$

Definition 3 (Parallel composition of protocols) *Let $P = (S_P, L_P, D_P, F_P, s_{0_P})$ and $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$. Let $C = L_P \bigcap L_Q$. The* ***parallel composition*** *between P and Q is defined as the LTS $P||Q = (S_P \times S_Q,\ L_P \cup L_Q,\ D,\ F_P \cup F_Q,\ (s_{0_P}, s_{0_Q}))$ where the transition relation D is defined as follows:*

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

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

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

3.2.2 Abstracting protocols to reason about functional matching

Given the definition of an extended LTS associated with the interaction protocols run by networked systems, we want to identify whether two protocols functionally match and, if so, to synthesize the mediator that enables them to interoperate, despite protocol-level mismatches.

With *functional matching* we mean that given two systems with respective interaction protocols P and Q, and ontologies O_P, O_Q describing their actions, *part of the behavior* of P and Q can synchronize. That is, a portion of the provided (required) functionalities of one protocol can synchronize with some required (provided) functionalities in the other, modulo an ontology mapping and a protocols' abstraction. Thus, we expect to find, at a given level of abstraction, similarities in the structure of the protocol representation of P and Q. This leads us to formally analyze such alike protocols to find, if it exists, a suitable mediator that allows the interoperability that otherwise would not be possible.

The definitions that follow allow reasoning about the appropriate structures of protocols. The first definition concerns states of the extended LTS from which at least two transitions start.

Definition 4 (Branch state) *Let $P = (S, L, D, F, s_0)$ and $s \in S$. s is a branch state, also written $branch(s)$, if $\exists B = \{d : d \in D$ and $d = (s, l, s')\}$ and $|B| \geq 2$.*

The second definition refers to states that identify the entry point of some cycles. That is: (i) there exists a trace that starts from and ends into a state s and (ii) there exists a transition (s_i, l, s), where l is not included in any cycling trace. Then s is called entry cycle state. An example of entry cycle state is in Figure 3.1(a).

Definition 5 (Entry cycle state) *Let $P = (S, L, D,\ F, s_0)$ and $s \in S$. s is an entry cycle state, also written $entry\_cycle(s)$, if $\exists\ (s_i, l, s) \in D$ for some $s_i \in S$ and for any $s \overset{t}{\Rightarrow} s$ it holds that $l \notin t$.*

Note that the length of a trace can also be 1, thus having a single transition in D having s as both starting and target state, that is $d = (s, l, s) \in D$.

The third definition identifies states of the extended LTS in which two or more transitions converge. An example of join state is in Figure 3.1(a).

Definition 6 (Join state) *Let $P = (S, L,\ D, F, s_0)$ and $s \in S$. s is a join state, also written $join(s)$, if $\exists\ J = \{d : d \in D$ and $d = (s_i, l, s)\}$ and $|J| \geq 2$ for some $s_i \in S$.*

The fourth definition generically defines as rich state any of the above defined states or an initial or final state. Examples of rich states are shown in Figure 3.1(a).

Definition 7 (Rich state) *Let $P = (S, L,\ D, F, s_0)$ and $s \in S$. s is a rich state, also written $rich(s)$, if it is either $branch(s)$ or $entry\_cycle(s)$, or $join(s)$, or $s = s_0$, or $s \in F$.*

Related to the previous definition is the notion of successive rich state. Given a rich state, the definition identifies the next immediately reachable rich state such that there is not any other rich state between them. An example is depicted in Figure 3.1(b).

Definition 8 (Successive rich state) *Let $P = (S, L, D, F, s_0)$ and rich(r)$\in S$. Successive rich state of r, also written $succ\_rich(s, r)$, is each $s \in S$ such that for any trace $r \overset{t}{\Rightarrow} s$ and $rich(s)$ it does not exist any other rich(s') between r and s.*

The *structure* of an extended LTS P follows from the previous definitions that introduce its building blocks. The set of states of the structure is the set of P's rich states. If there exists a trace $r \overset{t}{\Rightarrow} sr$ in P, then we say that a transition exists in the structure and it is labelled with t. Figure 3.3 shows two example of structures: one is that of the WM protocol of Figure 3.1(a) and the other is that of the JM protocol of Figure 3.1(b).

Definition 9 (Structure) *Let $P = (S, L, D, F, s_0)$. P', the structure of P also written structure(P', P), is the LTS $P' = (S', L', D', F', s'_0)$ where $F' = F$ and $s'_0 = s_0$, $S' = \{s \in S : rich(s)\}$, $L' = \{t \in L^* : s \overset{t}{\Rightarrow} r$ and $succ\_rich(r, s)\}$, and $D' = \{(s_i, t, s_j) : s_i, s_j \in S'$ and $t \in L'\}$.*

The following definitions allow reasoning about protocols to establish their functional matching. Given two LTS P, Q, and two ontologies O_P, O_Q describing their respective labels, the *common language* between P and Q identifies the actions of the protocols that have the same meaning and that form the basis for an interaction. The common language identification is based on the protocols' ontology mapping and on the correspondences between (one or sequences of) actions. It is made by a set of pairs of labels of O_P, O_Q such that the labels of $O_P(O_Q)$ are *mapped* [38] onto labels of $O_Q(O_P)$. We specialize the mapping definition by considering also pairs in which the elements are made by more than one label. More formally:
Let $P = (S_P,\ L_P, D_P, F_P, s_{0_P}), Q = (S_Q, L_Q,\ D_Q, F_Q, s_{0_Q})$. Let $O_P = (V_P, A_P), O_Q = (V_Q, A_Q)$ be ontologies with vocabularies $V_P = L_P^*$ and $V_Q = L_Q^*$, and such that their respective interpretations are specified by the sets of axioms A_P, A_Q. Let $maps : L_P^* \rightarrow L_Q^*$ be an **ontology mapping function**. Let $\alpha \in L_P^*, \beta \in L_Q^*$. We say that $\alpha(\beta)$ **corresponds** to $\beta(\alpha)$, also written $corresp(\alpha, \beta)$, if and only if it exists, through the ontology mapping, a splitting $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$, $\beta = \beta_1, \beta_2, \ldots, \beta_n$ such that $\forall\ 1 \leq i \leq n\ \exists j$ such that $\beta_i = maps(\overline{\alpha_j})$. We note that α and β can be of different size.

Definition 10 (Common Language) *Let $P = (S_P,\ L_P, D_P, F_P, s_{0_P}), Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$. Let $O_P = (L_P, A_P), O_Q = (L_Q, A_Q)$ be the ontologies of P, Q respectively. Let $maps : L_P^* \rightarrow L_Q^*$ be the ontology mapping function of P, Q respectively.*

The common language between P and Q, also written $common\_lang(P,\ Q)$, is the set $C = \{(\alpha, \beta) : corresp(\alpha, \beta)\}$.

In the messengers example, we have $C = \{((\overline{message}, ack), msg),\ ((message, \overline{ack}), \overline{msg})\}$.

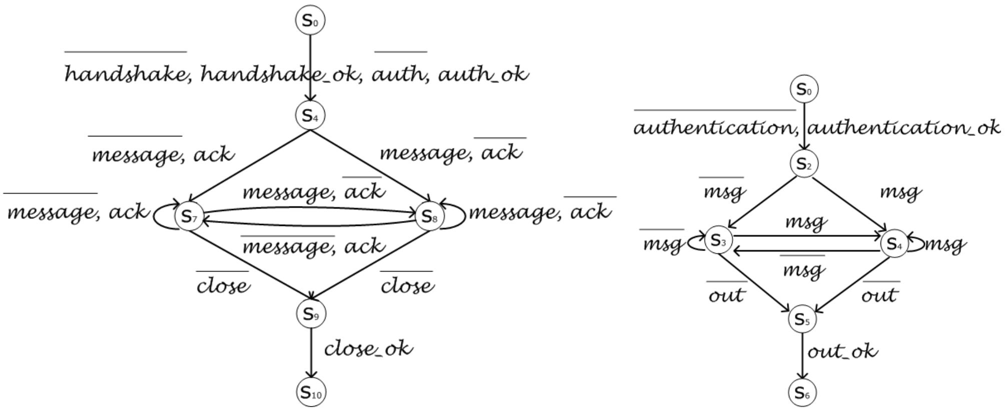
Definition 11 (Common Language Projected) *Let $P = (S_P, L_P, D_P, F_P, s_{0_P}), Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$. The common language projected on $P(Q)$ is the set $C_P(C_Q) = \{\alpha\ (\beta) : \exists\ (\alpha, \beta) \in common\_lang(P, Q)\ and\ \alpha \in L_P^*\ and\ \beta \in L_Q^*\}$.*

For example, the common language projected on WM protocol is the set $C_P = \{(\overline{message}, ack), (message, \overline{ack})\}$.

Definition 12 (Third Parties Language) *Let $P = (S_P, L_P, D_P, F_P, s_{0_P}), Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$. Let C_P, C_Q be the common language projected on P, Q respectively. The third parties language of $P(Q)$ is the set of words $T_P(T_Q)$ such that $T_P(T_Q) = \{\alpha \in L_P^*(L_Q^*)\ and\ \alpha \notin C_P(C_Q)\}$.*

Still considering the messengers example, let $S(S')$ be the protocol of the WM server(JM server). The third parties language of WM (JM), is the set $T_P = \{\ (\overline{handshake}), (handshake\_ok), (\overline{auth}), (auth\_ok), (\overline{close}), (close\_ok)\}$ ($T_Q = \{(\overline{authentication}), (authentication\_ok), (out), (out\_ok)\}$) that can synchronize with S protocol.

The following definition, starting from a LTS, builds its structure so that only labels belonging to the common language are preserved while the other labels are replaced by τ.



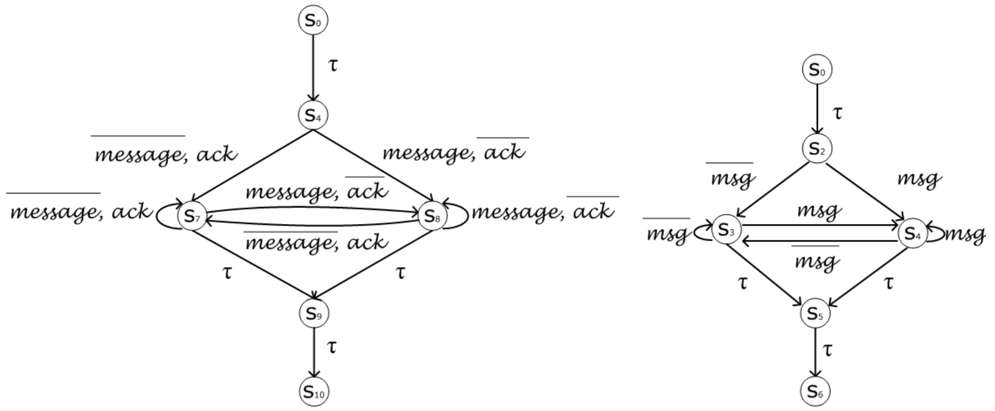
(a) Windows Messenger structure

(b) Jabber Messenger structure

Figure 3.3: Structures of the WM and JM protocols

Definition 13 (Induced LTS) *Let $P = (S, L, D, F, s_0)$. Let $L'^* \subseteq L^*$. The induced LTS of P by L'^* is the structure of P whose labels belonging to L'^* are observable while the others are replaced by τ. Sequences of τ within single transitions are replaced by only one τ.*

Examples of induced LTSs are shown in Figure 3.4.



(a) Induced LTS of WM protocol by the common language between WM and JM protocols projected on WM protocol

(b) Induced LTS of JM protocol by the common language between WM and JM protocols projected on JM protocol

Figure 3.4: Induced LTSs of the WM and JM protocols

3.2.3 Functional matching of protocols

The formalization described so far is needed to: (1) structurally characterize and (2) identify (if they exist) portions of protocols that can potentially interoperate. In order to establish if two protocols P, Q implement complementary functionalities and then to establish if there exists the possibility for them to interoperate, we use a suitable equivalence relation, the *functional matching relation*. Informally, this relation succeeds if for the part concerning the common language between P, Q, their control flow is bisimilar and there is a correspondence through an ontology mapping between their labels.

Definition 14 (Functional matching) *Let $P = (S_P, L_P, D_P, F_P, s_{0_P}), Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$ and let $s_p, s'_p \in S_P$, $s_q, s'_q \in S_Q$. Let $O_P = (L_P, A_P), O_Q = (L_Q, A_Q)$ be their respective ontologies. Let $maps : L^*_P \rightarrow L^*_Q$ be their ontology mapping function. Let C_P, C_Q be the common language projected on*

P, Q respectively. Let τ^* denote zero or more τ. Let $\alpha(\beta) \in L_P^*(L_Q^*)$, such that $\tau \neq \alpha(\beta)$. Let P' (Q') be P (Q) where for each $\alpha \in C_P$ ($\beta \in C_Q$), each label $\tau^*.\alpha.\tau^*$ in P ($\tau^*.\beta.\tau^*$ in Q) is replaced by α (β).
*P' has a **functional matching** to Q', also written $P' \simeq Q'$, iff the following conditions hold:*

i) $s_{0_P} \simeq s_{0_Q}$ holds by definition;

ii) if $s_p \simeq s_q$ and $\forall s_p \overset{\tau^*.\alpha.\tau^*}{\Longrightarrow} s'_p$ then $\exists s_q \overset{\tau^*.\beta.\tau^*}{\Longrightarrow} s'_q$ such that corresp(α, β) and $s'_p \simeq s'_q$;

iii) if $s_p \simeq s_q$ and $\forall s_q \overset{\tau^*.\beta.\tau^*}{\Longrightarrow} s'_q$ then $\exists s_p \overset{\tau^*.\alpha.\tau^*}{\Longrightarrow} s'_p$ such that corresp(α, β) and $s'_p \simeq s'_q$.

3.3 Towards Automated Matching and Synthesis

Building on the formalization of the previous section, we present the notions to establish if two protocols P and Q, inserted in a context C, are compatible. If so, we show how it is possible to synthesize a supporting mediator.

3.3.1 Mediated matching

The functional matching relation has a central role while looking for a mediated matching (i.e, protocols can interoperate through a CONNECTor) between two protocols P, Q. Indeed, it is based on the functional matching of the induced protocols of P, Q by their common language. That is, a mediated matching between P, Q exists if and only if an abstract portion of them has the same flow structure and corresponding actions. More formally:

Definition 15 (Mediated matching) *Let P and Q be two LTSs. Let $O_P = (L_P, A_P), O_Q = (L_Q, A_Q)$ be their respective ontologies. Let C_P and C_Q be the common language projected on P and Q respectively. Let I_P, I_Q be the induced LTSs by C_P and C_Q respectively. A mediated matching between P and Q exists iff I_P has a functional matching with I_Q.*

This definition expresses a necessary and sufficient condition that characterizes the existence of our mediator between two protocols.

3.3.2 Ontology-based functional matching

In order to check if a mediator between two behaviorally mismatched protocols P and Q exists, a mediated matching between them has to exist. Our framework checks this condition basing on an ontology mapping.

Let us consider that a mediated matching between P and Q exists. If a mediated matching exists, then a functional matching between the induced LTS of P (I_P) and of Q (I_Q) (abstractions of P and of Q respectively) has to exist. In order for a functional matching between I_P and I_Q to exist, the protocols P and Q needs to share a common language. To share a common language means that there exists an ontology mapping, between the languages of P and of Q, such that it equates at least a subset of the languages. It has to be noticed that for the portion of protocols labelled with labels belonging to the common language, a structural matching is implied by the functional matching relationship. For the remaining part (the one labelled with labels belonging to the third parties language) we assume that protocols synchronize with the context (e.g. servers in our messengers example). We are assuming that the unique mechanism to communicate is synchronization. That is a full synchronization between P and Q is achieved through the parallel composition $P||Q||C$ where the result of the parallel composition is that all actions belonging to the common language of P and Q are paired (send-receive) and all actions of the third parties language of P, Q are paired (send-receive) with some actions of the context. We recall that the portion of protocols labelled by the common language between P and Q are functionally similar and complementary. In other words every time that P performs an output action (belonging to the common language), there has to exists in Q (in its common language) the respective input action, and this implies the structural matching.

3.3.3 Abstract mediator synthesis

Let us recall that given two protocols P, Q such that there is a mediated matching between P and Q and a context C, we want to synthesize a mediator M such that the parallel composition $P||M||Q||C$, allows P, Q to evolve to their final states if any.

The actions of P, Q can belong to two sets: the *common language* and the *third parties language*. Based on this observation, we build the mediator as two separate components: COM and TH, if it exists. COM is an LTS built starting from the common language between P and Q, which aim is to solve the protocol-level mismatches occurring among complementary interactions (corresponding words). TH can be made by the parallel composition of two LTSs, if they exist, derived from P and from Q. The aim of TH is to forward the interactions between P, Q and their respective third parties.

The formal specification of the synthesis algorithm is presented in Figure 3.5.

```
Let  $P$          =  $(S_P, L_P, D_P, F_P, s_{0_P})$,
     $Q$          =  $(S_Q, L_Q, D_Q, F_Q, s_{0_Q})$
     $CONT$       =  $(S_{CONT}, L_{CONT}, D_{CONT}, F_{CONT}, s_{0_{CONT}})$.
     $O_P$        =  $(L_P^*, A_P)$   and
     $O_Q$        =  $(L_Q^*, A_Q)$   be the respective ontologies of $P$ and $Q$.
     $C$          =  $comm\_lang(P, Q)$.
     $C_P, C_Q$      be the common language projected on $P, Q$ respectively.
     $T_P, T_Q$      be the third parties language of $P, Q$ respectively.
The mediator $M$ is equal to $COM$ [ || $TH$]
where $COM$       =  $(S_C, L_C, D_C, F_C, s_{0_C})$ and
      $TH$        =  $(S_T, L_T, D_T, F_T, s_{0_T}) = [M_{T_P}]$ || [ $M_{T_Q}$].
$COM, TH$ are built as follows:
foreach $\alpha \in T_P(T_Q)$ do
  build a cycle made by subsequent transitions on the initial state of $M_{T_P}(M_{T_Q})$;
  insert a transition with label $\alpha$ followed by a transition with label $\overline{\alpha}$;
  foreach $\alpha \in C_P(C_Q)$ do
    if ($\alpha = \overline{\alpha}$), i.e., it is a send, then
      build a cycle made by subsequent transitions on the initial state of $COM$ ($s_{0_C}$);
      insert a transition with label $\alpha$ followed by a transition with label $\overline{\beta}$ where corresp($\overline{\alpha}, \beta$)
    else ($\alpha = \alpha$ i.e. it is a receive)
      build a cycle made by subsequent transitions on $s_{0_C}$;
      insert a transition with label $\beta$ followed by a transition with label $\overline{\alpha}$ where corresp($\overline{\beta}, \alpha$);
```

Figure 3.5: Synthesis algorithm

If L_{CONT}^*, the language of the context, contains the corresponding actions for both the third parties language of P, Q, if any, then the the parallel composition $P||Q||M||C$ let evolve P, Q to their final states.

3.4 Application of the Mediator Theory to the Popcorn Scenario

Considering the CONNECT overall challenge, we highlight here the specific contribution of our work on CONNECTor synthesis with respect to the overall CONNECT dynamic process. Specifically, we use the Popcorn scenario, also called Distributed Marketplace, introduced in [2] to illustrate the synthesis algorithm.

3.4.1 Heterogeneous merchant and consumer

Consider the case of a German consumer (i.e, implemented using Lime tuple space) and the French merchant (i.e, implemented using UPnP), see Figure 3.6. Figures 3.7 and 3.8 give the LTSs of the consumer behavior and of the merchant behavior respectively.

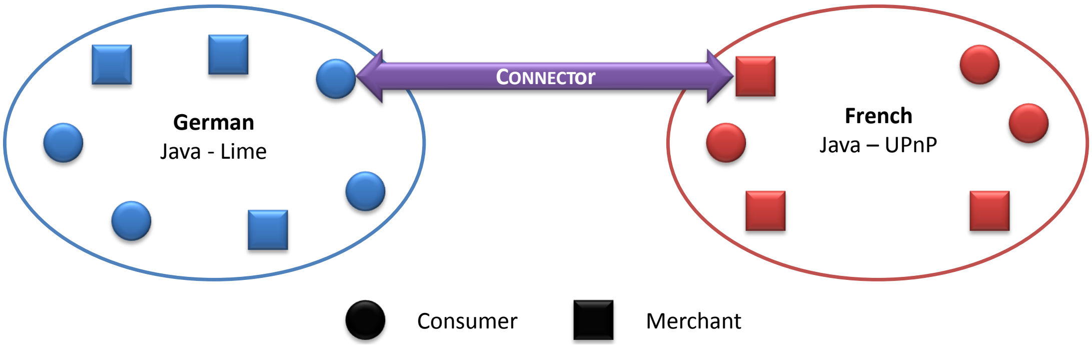


Figure 3.6: Popcorn scenario: German consumer - French merchant

Informally, the German consumer behaves as follows: he first browses the tuple space to retrieve the list of all merchants. Once he gets it, he looks for details about the merchants that sell a specific product (popcorn in our case) with a certain price (for example, less then a threshold) and some measure of distance (for example, within a given range).

Then, he writes into the tuple space a request to a chosen merchant of the product, also specifying the quantity and waits for a response. If everything is fine, the consumer will receive a positive response to the request and wait for a signal of proximity that will be sent by the merchant when he will be close to him. Otherwise, the consumer will receive a negative response (e.g, because the merchant has no sufficient quantity of product to satisfy the request). In both cases, the consumer can either restart from the beginning, i.e., from browsing the tuple space, or send a new request.

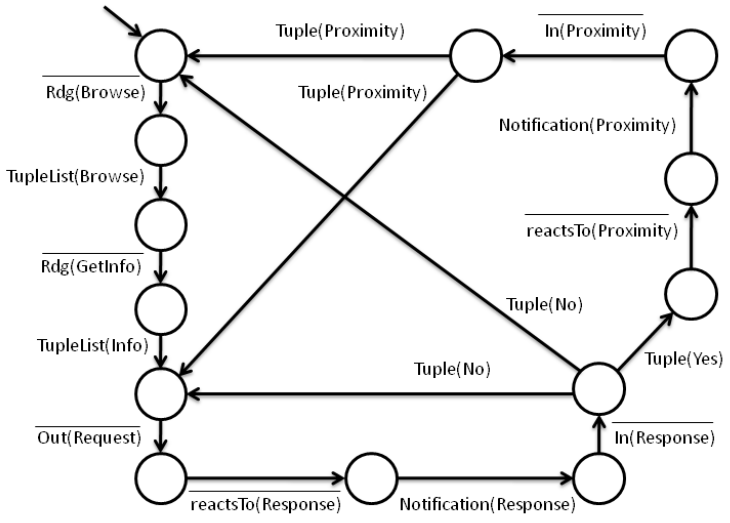


Figure 3.7: Tuple space consumer

The behavior of the merchant can be roughly described as follows: he receives queries from consumers and sends answers to them advertising his information. Then he receives more requests of information from the consumers and answers them providing the required information. Further he receives requests of ordering of products from the consumers and answers a consumer either: positively sending a proximity message when he is physically close to the consumer, or negatively in case he is not able to satisfy the request.

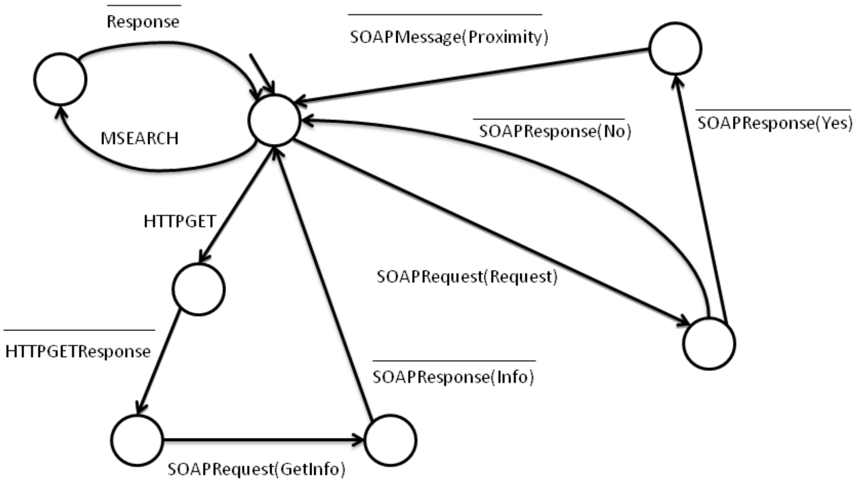


Figure 3.8: UPnP merchant

Even though these two applications have complementary behaviors, they are very different and they are not able to interoperate. Here the need for an appropriate interoperability solution clearly emerges.

The solution we are proposing is to synthesize a mediating connector that allows the merchant and consumer applications to interoperate. Specifically, with respect to the German consumer, the CONNECTor behaves as a compatible tuple space merchant. And, with respect to the French merchant, the CONNECTor behaves as a compatible UPnP consumer. Last but not the least, the CONNECTor makes suitable translations between enabled protocols.

The tuple space implementation of the Popcorn scenario is represented in Figure 3.9 and the portion that the mediating connector has to implement is the one with labels in red text. In more detail the behaviors of the tuple space that the connector should mimic are the following:

- It should receive and answer to browse requests about the merchants (namely, $Rdg(Browse)$ and $\overline{TupleList(Browse)}$);
- It should receive and reply to more detailed requests of information about the merchants (that is $Rdg(GetInfo)$ and $\overline{TupleList(Info)}$);
- It should receive specific ordering requests($Out(Request)$), receive the request to be informed when a response is available ($reactsTo(Response)$), notify that an answer is available and provide it ($\overline{Notification(Response)}$ and $In(Response)$ and either $\overline{Tuple(Yes)}$ or $\overline{Tuple(No)}$);
- It should receive the request to be informed when the merchant is in its vicinity, ($reactsTo(Proximity)$) and notify this fact with a message($\overline{Notification(Proximity)}$ and $In(Proximity)$ and $\overline{Tuple(Proximity)}$.

It has to be noticed that, in the first, third, and fourth steps, the mediator should also interact with the merchant in order to answer to the consumer.

Finally, by exploiting our knowledge about the consumer and merchant protocols, we extracted by hand the translation between the two different roles (consumer and merchant) as shown in Figure 3.10.

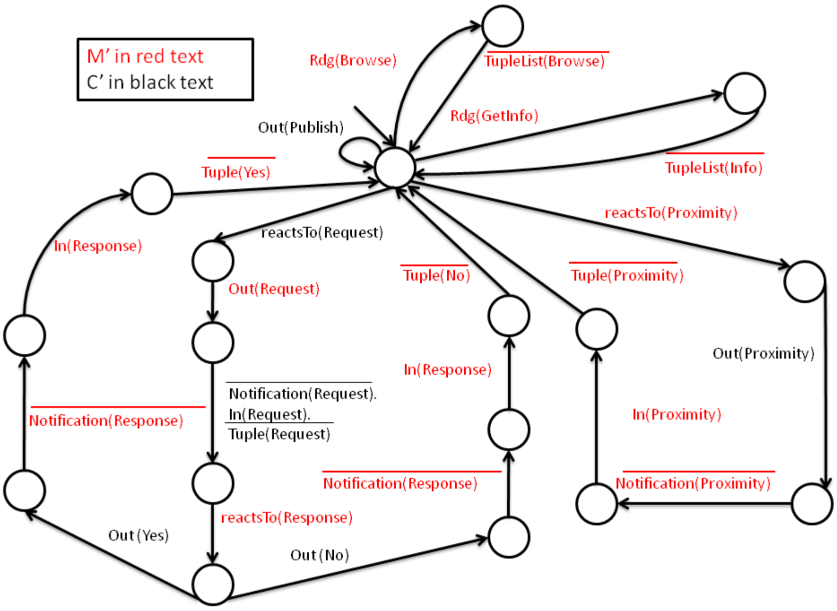
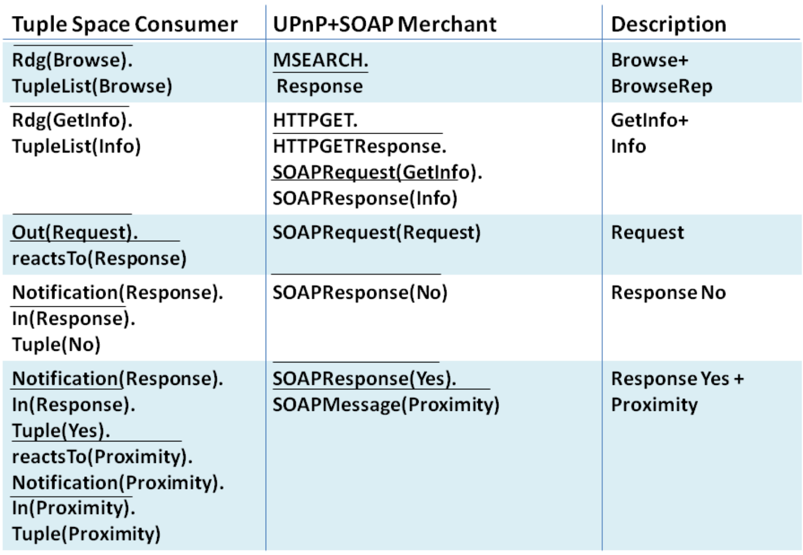


Figure 3.9: Tuple space implementation of the Popcorn scenario



Tuple Space Consumer	UPnP+SOAP Merchant	Description
Rdg(Browse). TupleList(Browse)	MSEARCH. Response	Browse+ BrowseRep
Rdg(GetInfo). TupleList(Info)	HTTPGET. HTTPGETResponse. SOAPRequest(GetInfo). SOAPResponse(Info)	GetInfo+ Info
Out(Request). reactsTo(Response)	SOAPRequest(Request)	Request
Notification(Response). In(Response). Tuple(No)	SOAPResponse(No)	Response No
Notification(Response). In(Response). Tuple(Yes). reactsTo(Proximity). Notification(Proximity). In(Proximity). Tuple(Proximity)	SOAPResponse(Yes). SOAPMessage(Proximity)	Response Yes + Proximity

Figure 3.10: Ontology mapping between tuple space consumer and UPnP Merchant

3.4.2 Applying mediated matching and mapping

Let us now analyze the application of our theory to the Popcorn scenario. We assume to have: the behavioral specification of consumer and merchant applications (as LTSs), their respective ontologies describing their actions, and the ontology mapping that defines the common language between consumer and merchant, i.e., represents their possible interactions. Indeed, as a first example we considered the messengers applications that concerns peer protocols and we designed suitable notions of structural and functional matching, based on *bisimulation*. Instead, this Popcorn scenario concerns client-server protocols and we modified the structural and functional matching definitions, based on *simulation*. It has to be noticed that further investigation are needed on the theory in order to highlight the relationship between protocol types and equivalence relation that has to be adopted.

With the application of a *slightly modified* version of the *theory* to the scenario, we obtained the CONNECTor of Figure 3.11. The building of this CONNECTor is driven by the behavior of the consumer, of the merchant and by their ontology mapping. In particular, between the consumer and the merchant's LTSs, the one with the "more restrictive behavior" is the consumer. Indeed, together with the merchant they reflect the client-server paradigm and they are in simulation relation, that is, the server simulates the client behavior. Hence, while building the CONNECTor's LTS, being driven by the synthesis algorithm, we have to be led by the the more restrictive behavior (consumer) and we have to create the appropriate sequences of actions always taking into account the ontology mapping.

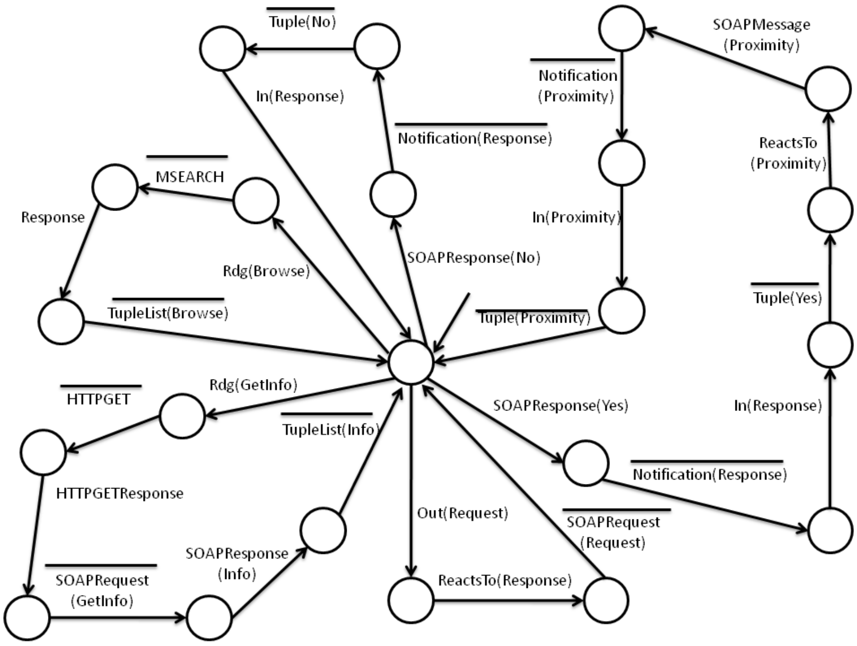


Figure 3.11: Mediating connector between tuple space consumer and UPnP merchant

3.5 Preliminary Assessment

In the previous sections, we have proposed a formal framework to precisely characterize interoperability between two networked systems that functionally match while having protocol mismatches. The purpose of the proposed formal model is to allow automated reasoning about functional matching and synthesis of the CONNECTors.

In the direction of an evaluation of the presented theory, we consider how *comprehensive* it is with respect to the *coverage of mismatches* that can occur during an interaction between two protocols. To achieve this aim, we consider the protocol mismatches classifications proposed in the area of Web services [23, 12, 42].

Basically, protocol-level mismatches occurring between functionally matching protocols are nothing but "behavioral problems" that prevent synchronization because of some "send/receive differences". There are six *basic mismatches* that can occur during a protocol interaction:

(1) *Extra send mismatch*: is about the send of actions from one protocol that have not the correspondent receive in the other protocol;

(2) *Extra receive mismatch*: one protocol does not issue a send action that the other is expecting to receive;

(3) *One send - many receive mismatches*: one protocol performs a sends action that corresponds to more than one receive action in the other protocol;

(4) *Many send - one receive mismatch*: concerns the send of more than one action by a protocol that correspond to only one receive action in the other protocol;

(5) *Signature mismatch*: the two protocols implement the same functionality/action but with different names;

(6) *Ordering mismatch*: one protocol performs a send action that the other is not expecting.

The mediator proposed in the previous sections, synthesized without the developers intervention, is able to detect and solve all the above described mismatches.

Let us consider two protocols and a context (third parties), such that the two protocols functionally match while mismatch behaviorally. Each of the protocols actions belongs to only one of two disjoint sets. These sets are the "common language" and the "third parties language". Further, a protocol action can at most be affected by a subset of the listed mismatches.

Intuitively, the algorithm identifies these mismatches thanks to the ontology mapping (plus the look ahead for the ordering mismatch case only). Instead, for solving the mismatches, the mediator does several things depending on the type of handled mismatch: it can translate and forward messages to the appropriate counterpart, or it can implement the complementary actions to synchronize with the ones that do not have the correspondence, or it can store and reorder actions.

Other aspects that will be interesting to investigate are complex mismatches that are obtained by mixing two or more of the basic ones listed above. Currently the synthesized mediator is able to deal with the complex mismatch obtained by combining all the basic mismatches (but (5)) with the signature mismatch. Note that in our approach solving mismatch (5) "is for free" thanks to the ontology mapping.

As we said before, what we presented in this section is mostly a discussion in the direction of an evaluation of the theory that we plan as future work together with the investigation of complex mismatches. In addition, a limit of the current approach is that we do not address data mismatches. We plan to investigate and extend our approach in this direction.

3.6 Summary

In this section, we briefly discuss the work we have been doing so far and the future work, in particular based on the requirements that emerged from the application of the theory for CONNECTors to the Popcorn scenario.

First of all, as we presented in the previous sections, we have formalized an initial theory [68, 67] for the mediating connectors that, as it is, is well suited for peer-to-peer protocols, i.e., protocols that implement the same "role" like the case of the two messenger clients. The need for changes arose while trying to apply this theory to the Popcorn scenario. In more detail, the concepts of *functional* and *structural matching* are central for establishing if the heterogeneous protocols are compatible and then if the possibility to communicate through a CONNECTOR exists for them. The definition of these concepts are dependent on the type of protocol considered. Indeed, while we adopted a modified version of bisimulation for the functional matching in the case of peer-to-peer protocols (messengers), this does not work for client-server protocols (consumer and merchant). In the latter case, we adopted a modified version of simulation. Also the definition of structure is bound to the type of protocol and needs to be changed accordingly. Hence, we assumed different meanings for the structure concept depending on the kind of protocols (i.e., peer, client, server, etc.) considered for the interoperability check.

With respect to the Popcorn scenario experiment, having in mind the whole CONNECT process [2], the synthesis enabler assumes to take as input, from the learning enabler[1], the behavior of the protocols to be made interoperable. At the end of the experiment, conducted independently by each partner, we found some differences between what we assumed as input from the learning enabler and what it is currently able to provide that till now does not seem to be severe. For example, in the learned LTS of the consumer there is only one transition labelled "Tuple(Proximity)" going back to the start state while in the LTS that we are assuming there also exists the possibility that leads the consumer to make another request. Furthermore, in the learned LTS of the merchant, there are two cycles to represent a request of information while we are assuming an LTS with only one loop.

Future work spans several directions including: the extension of the theory for CONNECTORs considering other types of protocols and the assessment of the theory thanks to the correctness proof. Furthermore, we will also investigate how to mitigate the gap between the behavioral models produced at the end of the learning phase and the ones assumed in our synthesis phase. Further future work concerns establishing whether we need a unified approach for both middleware- and application-layer mediation, or we should define two separate (possibly similar) approaches.

In the following we summarize the key points from this chapter, which relates to the assumptions underlying the proposed CONNECTor synthesis algorithm:

- The interaction protocols of networked systems are assumed to be modeled by means of LTSs whose transitions are labelled with the input/output messages that the systems exchange with their expected environment;

- Ontological information describing the meaning of the input/output actions of the protocols, and a mapping between ontologies of different protocols are provided.

With these assumptions, the CONNECTor synthesis algorithm is able to:

- Establish whether two protocols match and, hence, check whether there exists a mediator that lets the two protocol interoperate;

- If there exists such a mediator, produce a suitable protocol mapping; that is, automatically synthesize the required mediator.

[1] The protocol learning approach concerns the work conducted within work package WP4 [3].

4 Middleware-layer CONNECTor synthesis: Beyond State of the Art in Middleware Interoperability

In this chapter, we first highlight in Section 4.1 the different dimensions of interoperability to be addressed when concentrating on the the middleware-layer. Then, in Section 4.2, we recall some connector concepts in order to better understand the relation between middleware and connectors. In Section 4.3, we propose a formalization of the existing solutions to middleware-layer interoperability. Then, in Section 4.4 we consider the one based on dynamic protocol translation, as aimed by CONNECT and assess it through two different examples. In Section 4.5, we use the same two examples in order to evaluate the applicability at the middleware-layer of the approach to application-layer interoperability presented in Chapter 3. The conclusions and lessons learnt from these two experiences are discussed in Section 4.6, together with our future work.

4.1 Middleware Interoperability

Pervasive distributed systems often consist of many networked systems that are highly heterogeneous with respect to hardware, software and networks. These networked systems communicate via a plethora of disparate protocols leading to data and behavior incompatibilities. Solutions that dynamically reveal and fix interoperability issues are required to solve the mismatches that arise among the different running systems.

Interoperability can be considered from many perspectives and at different levels, from the application down to the network layer. Part of the focus should be on the middleware-layer since it stands as a conceptual paradigm to effectively connect heterogeneous systems. Moreover, application designers often choose a middleware first (based on the services provided), which may have an influence over the application since it implies the use of particular programming model.

Interoperable middleware have been introduced to overcome middleware heterogeneity. However, solutions remain rather static, requiring either the use of a proprietary interface or a priori implementation of protocol translators. In general, interoperability solutions solve protocol mismatches from application-layer down to middleware-layer at the syntactic level, which is too restrictive. This is particularly true when one considers the many dimensions of heterogeneity which arise in ubiquitous networking environments and require fine tuning of the middleware according to the specific capacities of the interacting parties. Thus, interoperable middleware can at best solve protocol mismatches that occur among domain-specific middleware. It is simply not possible to design beforehand a universal middleware solution that will enable effective networking of digital systems, while spanning the many dimensions of heterogeneity that currently exist in networked environments or which are likely to exist in the future. A revolutionary approach for the seamless networking of digital systems is to dynamically synthesize the connectors that make the networked systems able to communicate, as already presented in the previous chapter. This way neither the application nor the middleware itself need to be changed. And, since interactions in pervasive environments are generally spontaneous and dynamic, this adaptation should be fully automated.

Middleware provides the ability to dynamically find and use networked systems without any previous knowledge of their specific location/behavior. This purpose is achieved using discovery protocols. Several discovery protocols, like Jini [9], SLP [32]and SSDP [37] are now available, each of which is specific to a particular domain and has its own advantages and drawbacks. Once the networked systems are discovered, they need to interact using various paradigms, which have been classified into different architectural styles. The most important ones (see [8]) are: layered architectures, object-based architectures, data-centered architectures, and event-based architectures. Consequently, a second heterogeneity issue appears at the interaction level. Finally, non-functional properties, such as availability, reliability, timeliness or security, exhibited by the networked systems are frequently considered to be very important and are thus studied carefully. So, middleware should not only guarantee functional interoperability but also non-functional properties. Thus, in order to provide interoperability among middleware, three heterogeneity dimensions must be overcome: (i) discovery protocols, (ii) interaction protocols and, (iii) non-functional properties.

We intend to address such an issue through CONNECTor synthesis, in a way similar to our approach to application-layer protocol mediation. Toward this goal, we need to relate the middleware paradigm to that of connector.

4.2 Middleware-layer Connectors

In existing component models, connectors are meant to encapsulate interaction or communication while components are meant to encapsulate computation. In these models, control originates in components, and connectors are channels for coordinating the control flow (as well as data flow) between components [66]. In this section, we first briefly present the connector concept according to the software architecture work, which will help us to establish the relation between connector and middleware.

4.2.1 Connector definition

A software connector is defined in [75] as "*an architectural element tasked with effecting and regulating interactions among components*"

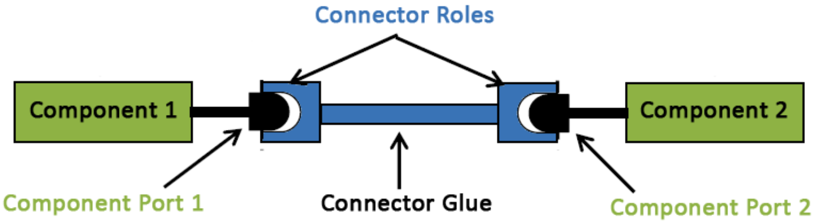


Figure 4.1: Component - Connector configuration

Formally, a connector type is defined by a set of *roles* and a *glue* specification (See Figure 4.1). The roles describe the expected local behavior of each of the interacting parties. The glue describes how the activities of these parties are coordinated [7]. Specifications for connectors are called protocols and described using process algebra, and in particular FSP [69]. The semantics of FSP is then expressed using LTS [45].

4.2.2 Connectors classification

There are many different kinds of connectors. The set is rich enough to require a taxonomy. We follow the one proposed in [75] and initially introduced in [48]. The classification framework includes: service category, type, dimension (and eventually subdimensions) and values for the dimensions (or subdimensions):

- *The service category* defines the interaction services the connector implements. There are four categories of interaction services:
  - *Communication* to support data transmission among components.
  - *Coordination* to support transfer of control among components.
  - *Conversion* to enable heterogeneous components to interact.
  - *Facilitation* to provide further mechanisms to facilitate or optimize the components interaction.
- *The type* describes the way the interaction services are realized.
  - *Procedure call connectors* use various invocation techniques and perform data transfer using parameters and return values. Thus, they provide both communication and coordination services. Examples of such connectors are CORBA remote procedure call, RMI, HTTP and SOAP.

  - *Event connectors* model the flow of control among components. Once an event (or an event pattern) happens, a message description (that is, event notification) is sent to all interested parties. Thus, they provide both communication and coordination services. Examples of such connectors are CORBA event channel service and JMS.
  - *Data Access connectors* allow components to access data maintained by a data store. Thus, they provide both communication and conversion services. Examples of such connectors are Linda and JavaSpaces.
  - *Linkage connectors* enable the establishment of communication and coordination channels between connectors that are then used by more functional connectors to enforce interaction semantics. Thus, they provide facilitation service. One example of such a connector is Service Binding.
  - *Stream connectors* perform the transfer of large amounts of data between autonomous processes. They provide communication service. Examples of such connectors are Unix pipe and TCP.
  - *Arbitrator connectors* streamline system operation, resolve any conflict and redirect the flow of control. They provide both coordination and facilitation services. Examples of such connectors are GLBP and transaction management systems.
  - *Adaptor connectors* provide facilities to support interaction between components that have not been designed to interoperate. They play a conversion role. Examples of such connectors are bridges like OrbixCOMet and SOAP2CORBA.
  - *Distributor connectors* perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths. They provide facilitation service. Examples of such connectors are DNS, routing protocols, and discovery protocols.

- *The dimensions and subdimensions* represent the architectural details of each connector type. For example, a procedure call connector has the following dimensions:
  - Parameters that are in turn subdivided to data trasfer, semantics, return value and invocation record.
  - Entry point which has two subdimensions, single or multiple.
  - Invocation which is implicit or explicit.
  - Synchronicity.
  - Cardinality that has two subdimensions, fan in and fan out.
  - Accessibility.

- *The values* represent the values a dimension or a subdimension can take. For example the data transfer subdimension can take the values reference, value or name.

4.2.3 Convergence of middleware and connector

Middleware facilitates communication and coordination of components that are distributed across several networked hosts. It provides a collection of services that take the primary responsibility of making distributed applications communicate. Middleware often also provides other services such as security, transaction, naming and events, which “aggregate” value to the communication between distributed applications [25].

From our perspective, middleware is represented by a set of connectors:

- *Discovery protocols* provide facilitation service of distributor type,

- *Interaction protocols* are represented by any connector type providing communication and coordination services, that is, procedure call, event, data access or stream connectors, and

- *Non-functional properties* can be modeled by the connector types providing facilitation services.

As presented in the next section, existing approaches to middleware interoperability primarily manage interoperability between connectors (middleware) of the same type, whereas we also aim at providing interoperability between connectors of different types within the same service category.

4.3 Formalizing Existing Approaches to Middleware Interoperability

Based on the work in [20], this section proposes an FSP-based formal specification of existing approaches to middleware interoperability. We briefly present each approach and focus on its formalization, which enables more accurate understanding and explanation of the approach than either an English language description or a reference implementation (more details about the implementations of these approaches can be found in [2]). It also provides a means to describe the approach in a way that it may be applied to different connector types. In addition, with a formal description it is possible to reason about connectors and help us to verify some properties and answer important questions about the effect of a particular approach. Relevant questions include: does it do what it claims (correctness)? Is it deadlock-free? Does it alter the interface of the communicating parties (transparency)?

Since we focus on connector behavior, it is natural to build on past work in this area, which uses process algebra that have proven to be the most adequate formalism to describe and reason about connector behavior. The process algebra chosen here is FSP because its notation and tool support were designed to be simpler to use than other process algebra, and it provides a useful set of analyzes such as safety and liveness verification.

4.3.1 FSP-based formalization

Based on the work in [69], a connector is formally defined in FSP [45] as a set of processes. Processes describe actions as events that occur in sequence and choices between event sequences. Each process P has an alphabet (αP) of the events that it is aware of. When composed in parallel, processes synchronize on *shared events* that is the events belonging to their respective alphabets. There is one process for each role of the connector, plus one process for the glue that describes how all the roles are bound together. These processes are placed in parallel with the roles relabelled. Figure 4.2 gives the semantics of a connector with roles $R_1 ... R_n$ and glue G.

$$\boxed{||Connector = R_1||R_2||...||R_n||G}$$

Figure 4.2: Connector specification

To illustrate this, consider a SOAP[1] (Simple Object Access Protocol) connector. It has two roles : $SOAP_{client}$ and $SOAP_{server}$. The $SOAP_{client}$ initiates a request, represented as a $cSOAP_{req}$ event, and get a response, represented as a $cSOAP_{resp}$ event. When the $SOAP_{server}$ observes a request $sSOAP_{req}$, it initiates a response $sSOAP_{resp}$. The $Glue_{SOAP}$ coordinates the interaction of the two roles: a $cSOAP_{req}$ from the $SOAP_{client}$is followed by an $sSOAP_{req}$ to the $SOAP_{server}$, and an $sSOAP_{resp}$ from the $SOAP_{server}$ is followed by a $cSOAP_{resp}$ to the $SOAP_{client}$ (See Figure 4.3).

On the other hand, a component may have multiple interfaces, each of which is termed a port. A port identifies a point of interaction between the component and its environment. Component ports are also specified by processes. Then, a component, represented by ports $P_1 ... P_n$, is attached to a connector, represented by roles $R_1 ... R_n$ and glue G by replacing each component port with a connector role. The replacement is possible if the component port is *compatible* with the connector role [29].

Using this connector specification, we propose a formal specification of existing solutions to middleware interoperability. As in [2], we consider several families of solutions: bridging, interoperability platforms and transparent interoperability (that also includes logical mobility as a special case).

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

//*SOAP* `Connector` *specification*		
Role $SOAP_{client}$	$=$	$cSOAP_{req} \rightarrow cSOAP_{resp} \rightarrow SOAP_{client}$
Role $SOAP_{server}$	$=$	$sSOAP_{req} \rightarrow sSOAP_{resp} \rightarrow SOAP_{server}$
Glue$_{SOAP}$	$=$	$cSOAP_{req} \rightarrow sSOAP_{req} \rightarrow Glue_{SOAP}$
	$\mid$	$sSOAP_{resp} \rightarrow cSOAP_{resp} \rightarrow Glue_{SOAP}$
$\Vert Connector_{SOAP}$	$=$	$SOAP_{client} \Vert Glue_{SOAP} \Vert SOAP_{server}$

Figure 4.3: SOAP connector specification

4.3.2 Bridging

Bridging assumes a priori knowledge of middleware (connectors) that have to interoperate without code modification and provides a mapping between various protocols. This mapping can either be $1 \rightarrow 1$, which is *direct bridging*; or $n \rightarrow 1 \rightarrow m$, which is *indirect bridging*.

Direct Bridging

The principle is to transform one of the connector roles according to the component port (see Figure 4.4). Formally, the glue of each connector is first tagged in order to avoid unwanted event synchronization (tag_1 :Glue$_1$ and tag_2 :Glue$_2$). Then, a set of transformations is applied to the connectors in order to adapt their respective behaviors (T). Finally, the transformations are chained with the glues through the Bridge process.

//*Specification of the* `connector`$_1$ *&* `connector`$_2$		
Role $R1_{i,i \in [1..2]}$	$=$	*Specification of Role* $R1$ *of* `connector`$_i$
Role $R2_{i,i \in [1..2]}$	$=$	*Specification of Role* $R2$ *of* `connector`$_i$
Glue$_{i,i \in [1..2]}$	$=$	*Specification of the glue of* `connector`$_i$
Set $I_{i,i \in [1..2]}$	$=$	*Set of events initiated from* `role` $R1_i$ and $R2_i$
Bridge	$=$	$tag_1.[e_1 : I_1] \rightarrow tag_2.[e_1]$ →Bridge
	$\mid$	$tag_2.[e_2 : I_2] \rightarrow tag_1.[e_2]$ →Bridge
//*Adaptation process*		
T	$=$	*Specification of the required transformations to bridge* `Connector`$_1$ *to* `Connector`$_2$
//*Semantic of the connector*		
$\Vert$C-DBridge	$=$	$R1_1 \Vert tag_1$:Glue$_1 \Vert$Bridge $\Vert$T$\Vert tag_2$:Glue$_2 \Vert R2_2$

Figure 4.4: Direct bridging specification

Direct bridges, such as OrbixCOMet[2] and SOAP2CORBA[3], provide interoperability between two fixed protocols (DCOM-CORBA and SOAP-CORBA respectively). A direct bridge must thus be developed separately for every pair of protocols between which interaction is needed. The diversity of protocols that are used in today's networked systems implies that this is a substantial development task.

Indirect bridging

Resolving heterogeneity among two sets of n and m middleware requires $n \times m$ direct bridges. An alternative approach is then to use a common fixed intermediary protocol. In this case, interoperability is achieved in two steps: first one native middleware protocol taken among n middleware is translated to a common intermediary protocol, then this is translated to another native middleware protocol taken among m middleware (see Figure 4.5). First, one of the n (m) connectors is selected using the Switch (Switch′) process: `Connector`$_i$ (`Connector`$'_k$). Then, direct bridges are used between `Connector`$_i$ and `Connector`$_{bus}$ ($ToT_i \Vert Bridge_i$), and between `Connector`$_{bus}$ and `Connector`$'_k$ ($ToT'_k \Vert Bridge'_k$).

Indirect bridges, such as Enterprise Service Buses (ESBs) or MUSDAC [61] rely either on an intermediary infrastructure or on a single fixed intermediary protocol they translate messages to and from it. This

[2] http://www.iona.com/support/whitepapers/ocomet-wp.pdf

[3] http://soap2corba.sourceforge.net/

//*Bus* `Connector`		
Role $R1_{bus}$	$=$	*Specification of Role* $R1$ *of* $\texttt{connector}_{bus}$
Role $R2_{bus}$	$=$	*Specification of Role* $R2$ *of* $\texttt{connector}_{bus}$
Glue$_{bus}$	$=$	*Specification that describes interactions between roles* $R1_{bus}$ *and* $R2_{bus}$
//`Connectors` *specification*		
Role $R1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow R1_i),$
$R1_{i,i\in[1\cdot\cdot n]}$	$=$	$R1_i$ *initial specification as given by* $\texttt{Connector}_i$
	$\vert$	$reset \rightarrow$ R1
Role $R2$	$=$	$\vert_{k=1}^{m}(b.glue'_k \rightarrow R2_k),$
$R2_{k,k\in[1\cdot\cdot m]}$	$=$	$R2_k$ *initial specification as given by* $\texttt{Connector}'_k$
	$\vert$	$reset \rightarrow$ R2
Glue$_{i,i\in[1\cdot\cdot n]}$	$=$	*Specification that describes interactions between roles* $R1_i$ *and* $R2_i$
Glue$'_{k,k\in[1\cdot\cdot m]}$	$=$	*Specification that describes interactions between roles* $R'1_k$ *and* $R'2_k$
//*Set of events initiated or observed*		
Set $I1_{i,i\in[1\cdot\cdot n]}$	$=$	*Set of events initiated from* `role` $R1_i$
Set $O1_{i,i\in[1\cdot\cdot n]}$	$=$	*Set of events observed from* `role` $R1_i$
Set $I2_{k,k\in[1\cdot\cdot m]}$	$=$	*Set of events initiated from* `role` $R'2_k$
Set $O2_{k,k\in[1\cdot\cdot m]}$	$=$	*Set of events observed from* `role` $R'2_k$
//*Switch processes*		
Switch	$=$	$(a.election \rightarrow a.reset \rightarrow$ Switch
	$\vert_{i=1}^{n}$	$a.election \rightarrow a.glue_i \rightarrow$ Switch$)\backslash\{a.election\}$
Switch$'$	$=$	$(b.election \rightarrow b.reset \rightarrow$ Switch$'$
	$\vert_{k=1}^{m}$	$b.election \rightarrow b.glue'_k \rightarrow$ Switch$')\backslash\{b.election\}$
//*Adaptation processes*		
T$_1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow ToT_i),$
$ToT_{i,i\in[1\cdot\cdot n]}$	$=$	*Specification of the required transformations to bridge* $\texttt{Connector}_i$ *to* $\texttt{Connector}_{bus}$
	$\vert$	$a.reset \rightarrow$T$_1$
T$_2$	$=$	$\vert_{k=1}^{m}(b.glue'_k \rightarrow ToT'_k),$
$ToT'_{k,k\in[1\cdot\cdot m]}$	$=$	*Specification of the required transformations to bridge* $\texttt{Connector}_{bus}$ *to* $\texttt{Connector}'_k$
	$\vert$	$b.reset \rightarrow$T$_2$
//*Bridging processes*		
Bridge$_1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow Bridge_i),$
$Bridge_{i,i\in[1\cdot\cdot n]}$	$=$	$[e : I1_i] \rightarrow a.tag_i.[e] \rightarrow Bridge_i$
	$\vert$	$a.tag_i.[e : O1_i] \rightarrow [e] \rightarrow Bridge_i$
	$\vert$	$a.reset \rightarrow$Bridge$_1$
Bridge$_2$	$=$	$\vert_{k=1}^{m}(b.glue'_k \rightarrow Bridge'_k),$
$Bridge'_{k,k\in[1\cdot\cdot m]}$	$=$	$[e : I2_k] \rightarrow b.tag_k.[e] \rightarrow Bridge'_k$
	$\vert$	$b.tag_i.[e : O2_k] \rightarrow [e] \rightarrow Bridge'_k$
	$\vert$	$b.reset \rightarrow$ Bridge$_2$
//*The* `Connector`		
$\Vert$C-IBridge	$=$	$R1\Vert$Switch$\Vert$T$_1\Vert_{i=1}^{n}a.tag_i$:Glue$_i$ $\Vert$Bridge$_1\Vert$Glue$_{bus}\Vert$Bridge$_2$ $\Vert_{k=1}^{m}b.tag_k$:Glue$'_k\Vert$T$_2\Vert$Switch$'\Vert R2$

Figure 4.5: Indirect bridging specification

approach reduces the development effort from n^2 to $n + m$, but may limit the expressiveness, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol.

4.3.3 Interoperability platforms

To overcome the static nature of bridges, approaches that dynamically select the best middleware bridge at a given time and place have emerged. Interoperability platforms enable clients or services to switch their interaction protocol on-the-fly according to their networked environment. The principle is to provide an explicit interface that abstracts the different interaction protocols used in the environment (see Figure 4.6). The interface is formally specified by a role $R_{interface}$. A non-deterministic process (Switch) selects the appropriate connector among n: $\texttt{Connector}_i$. Then, the $R_{interface}$ is bridged to $\texttt{Connector}_i$ using the same method than direct bridging, that is $ToT_i \| Bridge_i$.

$$
\begin{array}{lcl}
\textit{//Proprietary interface} & & \\
\text{Role } R_{interface} & = & \textit{Specification of the bridge interface} \\
\text{Role } R2 & = & |_{i=1}^{n} (glue_i \rightarrow R2_i), \\
R2_{i,i \in [1 \cdot\cdot n]} & = & \textit{Initial specification of the role } R2 \textit{ of } \texttt{Connector}_i \\
 & | & reset \rightarrow \text{R2} \\
\text{Glue}_{i,i \in [1 \cdot\cdot n]} & = & \textit{Specification of the glue of } \texttt{Connector}_i \\
\textit{//Set of events initiated or observed} & & \\
\text{Set } I2_{i,i \in [1 \cdot\cdot n]} & = & \textit{Set of events initiated from } \texttt{role} \;\; R2_i \\
\text{Set } O2_{i,i \in [1 \cdot\cdot n]} & = & \textit{Set of events observed from } \texttt{role} \;\; R2_i \\
\text{Set } I_{interface} & = & \textit{Set of events initiated from } \texttt{role} \;\; R_{interface} \\
\text{Set } O_{interface} & = & \textit{Set of events observed from } \texttt{role} \;\; R_{interface} \\
\textit{//Switch process} & & \\
\text{Switch} & = & (election \rightarrow reset \rightarrow \text{Switch} \\
 & |_{i=1}^{n} & election \rightarrow glue_i \rightarrow \text{Switch}) \backslash \{election\} \\
\textit{//Adaptation process} & & \\
\text{T} & = & |_{i=1}^{n} (glue_i \rightarrow ToT_i), \\
ToT_{i,i \in [1 \cdot\cdot n]} & = & \textit{Specification of the required transformations to bridge} \\
 & & R_{interface} \;\textit{ to } \texttt{Connector}_i \\
 & | & reset \rightarrow \text{T} \\
\textit{//Bridging process} & & \\
\text{Bridge} & = & |_{i=1}^{n} (glue_i \rightarrow Bridge_i), \\
Bridge_{i,i \in [1 \cdot\cdot n]} & = & [e : R_{interface}] \rightarrow tag_i.[e] \rightarrow Bridge_i \\
 & | & tag_i.[e : O_{interface}] \rightarrow [e] \rightarrow Bridge_i \\
 & | & [e : I2_i] \rightarrow tag_i.[e] \rightarrow Bridge_i \\
 & | & tag_i.[e : O2_i] \rightarrow [e] \rightarrow Bridge_i \\
 & | & reset \rightarrow \text{Bridge} \\
\textit{//The } \texttt{Connector} & & \\
\|\text{C-InteropPlatforms} & = & R_{interface} \| \text{Switch} \| \text{T} \| \text{Bridge} \|_{i=1}^{n} tag_i : \text{Glue}_i \| R_2
\end{array}
$$

Figure 4.6: Interoperability platforms specification

Interoperability platforms such as UIC [62] and ReMMoC [31], allow the development of applications independently from the underlying protocol. They select the most appropriate communication protocol according to the context. Many applications, however, have not been developed using such middleware systems and cannot be modified because their source code, for example, is not available.

4.3.4 Transparent interoperability

Transparent interoperability solutions do not rely on a fixed common protocol anymore but rather synthesize it dynamically based on the interaction behavior of communicating parties in a way similar to the CONNECT approach to the synthesis of mediating connector discussed in Chapter 3.

In this deliverable, we are more specifically interested in *dynamic protocol translation* [18]. This approach extends indirect bridging solutions with concepts taken from the theory of protocol projection [41]. The theory enables mapping incompatible protocols to an image protocol, which proves useful for reasoning about conversions and semantic equivalence among heterogeneous protocols [18]. In particular, an image protocol abstracts incompatibilities among protocols to exclusively consider their similarities. Further, by generating an image protocol on-the-fly, it is possible to provide a dynamic semantical correspondence among heterogeneous middleware protocols. In other terms, a projection function f is used to synthesize an image protocol resulting from the greatest common denominator of the different middleware protocol similarities (see Figure 4.7). First, the glue of all the connectors are tagged in order to avoid unwanted event synchronization. Then, one connector is chosen among n (m) connectors through the Switch (Switch′) process: $\texttt{Connector}_i$ ($\texttt{Connector}'_k$). W$_1$ (W$_2$) are then used to synchronize tagged glues with their respective roles depending on the selected connector. The strength of the approach lies in M$_1$ and M$_2$ processes that are used to define the semantics of the events. To do so, a projection function (f) is used to establish the semantic equivalence between events: $f(e_1) = f(e_2)$ if and only if e_1 and e_2 have the same semantics. Finally, Bridge$_1$ and Bridge$_2$ tag/untag the projected events in order to allow M$_1$ and M$_2$ to synchronize. Thus, the approach is fully automated, the only requirement is the definition of the semantics of events using the f function.

The INDISS [19] and NEMESYS [20] middleware implement the dynamic protocol translation approach for service discovery and interaction protocol, respectively. uMiddle [58], OSDA [43], SeDiM [27] are other implementations of the transparent interoperability approach.

4.4 Assessing the Transparent Interoperability Approach

To better understand the transparent interoperability approach, and in particular the one described in [18], consider the Popcorn scenario that is detailed in [2] and already used in Section 3.4.

As stated in Section 4.1, interoperability issues at the middleware-layer may arise due to three dimensions: discovery, interaction and non-functional properties. Note that the non-functional dimension will be addressed in future work. Since the different actors (Popcorn Merchants and Consumers) do not know each other beforehand, they have to locate and discover each other at runtime. Middleware support for service discovery is indispensable for developing applications in these highly heterogeneous environments. As mentioned before, different discovery protocols are deployed in today's networked environments. For our study, we have chosen Service Location Protocol (SLP) [32] and Simple Service Discovery Protocol (SSDP) [37] since they are among the most broadly used dynamic service discovery protocols (Section 4.4.1). This further illustrates interoperability among connectors of the same type but with heterogeneous dimensions.

Then, we address a more complex and not yet addressed case (as the best of our knowledge) that is considering interoperability among different connector types of the same category. We more specifically investigate interoperabilty among Universal Plug and Play(UPnP) [37] and Linda in a Mobile environment (Lime) [57]. Thus, we will be dealing with different connector types: distributor-procedure call connectors for UPnP, and data access-event for Lime (Section 4.4.2).

4.4.1 Example 1: Interoperability within the same connector type

Service Discovery interoperability has been widely addressed and many approaches have been proposed. We consider the transparent interoperability approach, as defined in [18], since it efficiently addresses interoperability at runtime between a set of components, as aimed by CONNECT. To do so, a set of semantic events is associated with any discovery protocol, which helps in defining the *image protocol*, that is the intermediary protocol, in order to use the transparent interoperability approach to make SLP interoperate with SSDP , that is the Spanish/French Popcorn scenarios described in [2]. We first start by describing and formalizing SSDP and SLP. Then, we show how to apply the approach to this example.

Simple Service Discovery Protocol(SSDP)

The Simple Service Discovery Protocol (SSDP) is the Universal Plug and Play (UPnP) discovery protocol. UPnP defines two network entities:

- Devices: implement the protocols required by the UPnP architecture.
- Control points: ask for a functionality provided by a device.

The UPnP control points and devices represent one multicast group using the IP address 239.255.255.250 and port 1900. Devices advertise the services that are providing using *alive* messages. Control points look for services by multicasting a *MSEARCH* message. The devices reply by sending a unicast *response*

//`Connectors` *specification*		
Role $R1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow R1_i),$
$R1_{i,i\in[1\cdot\cdot n]}$	$=$	$R1_i$ *Initial specification as given by* $\texttt{Connector}_i$
	$\mid$	$reset \rightarrow$ R1
Role $R2$	$=$	$\vert_{k=1}^{n}(b.glue_k \rightarrow R2_k),$
$R2_{k,k\in[1\cdot\cdot n]}$	$=$	$R2_k$ *Initial specification as given by* $\texttt{Connector}'_k$
	$\mid$	$reset \rightarrow$ R2
Glue$_{i,i\in[1\cdot\cdot n]}$	$=$	*Specification that describes interactions between roles* $R1_i$ *and* $R2_i$
Glue$'_{k,k\in[1\cdot\cdot m]}$	$=$	*specification that describes interactions between roles* $R'1_k$ *and* $R'2_k$
//*Definition of set of events*		
Set $I1_{i,i\in[1\cdot\cdot n]}$	$=$	*Set of events initiated from* `role` $R1_i$
Set $O1_{i,i\in[1\cdot\cdot n]}$	$=$	*Set of events observed from* `role` $R1_i$
Set $I2_{k,k\in[1\cdot\cdot m]}$	$=$	*Set of events initiated from* `role` $R'2_k$
Set $O2_{k,k\in[1\cdot\cdot m]}$	$=$	*Set of events observed from* `role` $R'2_k$
Set E1$_{i,i\in[1\cdot\cdot n]}$	$=$	αR1$_i \cap \alpha$Glue$_i$
Set E2$_{k,k\in[1\cdot\cdot m]}$	$=$	αR2$_k \cap \alpha$Glue$'_k$
Set $\sum_{E1_n}$	$=$	$\cup_{i=1}^{n} E1_i$
Set $\sum_{E2_m}$	$=$	$\cup_{k=1}^{m} E2_k$
Set $\sum_{O1_n}$	$=$	$\cup_{i=1}^{n} O1_i$
Set $\sum_{O2_m}$	$=$	$\cup_{k=1}^{m} O2_k$
//*Switch processes*		
Switch	$=$	$(a.election \rightarrow a.reset \rightarrow$ Switch
	$\vert_{i=1}^{n}$	$a.election \rightarrow a.glue_i \rightarrow$ Switch$)\backslash\{a.election\}$
Switch$'$	$=$	$(b.election \rightarrow b.reset \rightarrow$ Switch$'$
	$\vert_{k=1}^{m}$	$b.election \rightarrow b.glue'_k \rightarrow$ Switch$')\backslash\{b.election\}$
//*Image protocol generation*		
W$_1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow ToGlue_i),$
$ToGlue_{i,i\in[1\cdot\cdot n]}$	$=$	$[e : I1_i] \rightarrow a.tag_i.[e] \rightarrow ToGlue_i$
	$\mid$	$a.tag_i.[e : O1_i] \rightarrow [e] \rightarrow ToGlue_i$
	$\mid$	$a.reset \rightarrow$W$_1$
W$_2$	$=$	$\vert_{k=1}^{n}(b.glue'_k \rightarrow ToGlue'_k),$
$ToGlue'_{k,k\in[1\cdot\cdot m]}$	$=$	$[e : I2_k] \rightarrow b.tag_k.[e] \rightarrow ToGlue'_k$
	$\mid$	$b.tag_k.[e : O2_k] \rightarrow [e] \rightarrow ToGlue'_k$
	$\mid$	$b.reset \rightarrow$W$_2$
M$_1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow ToMap_i),$
$ToMap_{i,i\in[1\cdot\cdot n]}$	$=$	$a.tag_i.[e : I1_i] \rightarrow a.tag_i.f(e) \rightarrow ToMap_i$
	$\mid$	$a.tag_i.f(e : \sum_{[O1_n]}) \rightarrow a.tag_i.[e : O1_i] \rightarrow ToMap_i$
	$\mid$	$a.reset \rightarrow$M$_1$
M$_2$	$=$	$\vert_{k=1}^{n}(b.glue'_k \rightarrow ToMap'_k),$
$ToMap'_{k,k\in[1\cdot\cdot m]}$	$=$	$b.tag_k.[e : I2_k] \rightarrow b.tag_k.f(e) \rightarrow ToMap'_k$
	$\mid$	$b.tag_k.f(e : \sum_{[O2_m]}) \rightarrow b.tag_k.[e : O2_k] \rightarrow ToMap'_k$
	$\mid$	$b.reset \rightarrow$M$_2$
//*Bridging*		
Bridge$_1$	$=$	$\vert_{i=1}^{n}(a.glue_i \rightarrow ToBridge_i),$
$ToBridge_{i,i\in[1\cdot\cdot n]}$	$=$	$a.tag_i.f\left(e_2 : \sum_{[E2_k]}\right) \rightarrow f(e_2) \rightarrow ToBridge_i$
	$\mid$	$f(e_1 : \sum_{[E1_n]}) \rightarrow a.tag_i.f(e_1) \rightarrow ToBridge_i$
	$\mid$	$a.reset \rightarrow$Bridge$_1$
Bridge$_2$	$=$	$\vert_{k=1}^{m}(b.glue'_k \rightarrow ToBridge'_k),$
$ToBridge'_{k,k\in[1\cdot\cdot m]}$	$=$	$b.tag_k.f\left(e_1 : \sum_{[E1_n]}\right) \rightarrow f(e_1) \rightarrow ToBridge'_k$
	$\mid$	$f(e_2 : \sum_{[E2_m]}) \rightarrow b.tag_k.f(e_2) \rightarrow ToBridge'_k$
	$\mid$	$b.reset \rightarrow$Bridge$_2$
//*The* `Connector`		
C-Transparent_Interop	$=$	$R1\Vert$ Switch $\Vert_{i=1}^{n} a.tag_i$: Glue$_i/\{f(r : \alpha$Glue$_i)/[r]\}$ $\Vert$ W $_1\Vert$ M $_1\Vert$Bridge$_1$ $\Vert$Bridge$_2\Vert$M $_2\Vert$ W $_2$ $\Vert_{k=1}^{m} b.tag_k$: Glue$_k/\{f(r : \alpha$Glue$_k)/[r]\}\Vert$Switch $'\Vert R2$

Figure 4.7: Transparent interoperability specification

containing a device description. The control points process the device description and perform HTTP GET requests to get the corresponding services description. FSP-based specification of SSDP is illustrated in Figure 4.8.

```
UPNP_GROUP_MANAGER      =   IDLE,
IDLE                    =   (join[upnp] → MATCH
                        |   send[Multicast_groups] → IDLE
                        )   ,
MATCH                   =
                        (   send[group : Multicast_groups] →
                              if (group == upnp) then
                                (send[group] → MATCH
                                |leave → IDLE)
                              else
                                MATCH
                        |   leave → IDLE
                        )
UPNP_RECEIVER           =   (join[upnp] → LISTENING),
LISTENING               =   (send[upnp] → LISTENING
                        |   send[upnp] → response → LISTENING
                        |   send[upnp].alive[i : DeviceRange] → description[i] → LISTENING
                        |   leave → STOP
                        )   +{join[Multicast_groups]}
//Service description
UPNP_SERVICE            =   (httpget → httpgetResponse →UPNP_SERVICE)
//The join then advertise step
UPNP_JOIN_ADVERTISE     =   (join[upnp] → alive →STOP)/{alive/send[upnp]}
//Discovery step
UPNP_DISCOVERY          =   (msearch →UPNP_DISCOVERY
                        |   response[i : DeviceRange] → description[i] →UPNP_DISCOVERY
                        |   response[i : DeviceRange] →UPNP_DISCOVERY
                        )   /{msearch/send[upnp]}
//Description step
UPNP_DESCRIPTION        =   (description → httpget → httpgetResponse → UPNP_DESCRIPTION
                        |   description → httpget → httpgetResponse → control → UPNP_DESCRIPTION)
||SSDP                  =   (device[DeviceRange] :UPNP_GROUP_MANAGER
                        ||  device[DeviceRange] :UPNP_RECEIVER
                        ||  service[i : DeviceRange][j : ServiceRangei] :UPNP_SERVICE
                        ||  device[DeviceRange] :UPNP_JOIN_ADVERTISE
                        ||  device[DeviceRange] :UPNP_DISCOVERY
                        ||  service[i : DeviceRange][j : ServiceRangei] :UPNP_DESCRIPTION
                        )   /{description[i : DeviceRange]/service[i][ServiceRangei].description}
```

Figure 4.8: SSDP specification

Service Location Protocol (SLP)

The Service Location Protocol is an IETF standard that provides a scalable framework for automatic resource discovery on IP networks [32]. It includes three "agents" that operate on behalf of the network-based software:

- User Agents (UA) perform service discovery.
- Service Agents (SA) advertise the location and attributes of services.
- Directory Agents (DA) aggregate service information into what is initially a stateless repository. When a DA is present, it collects all service information advertised by SAs, and UAs unicast their requests to the DA. In the absence of a DA, UAs repeatedly multicast the same request they would have unicast to a DA. SAs listen for these multicast requests and unicast responses to the UA if it has advertised the requested service.

The SLP agents represent one multicast group using the IP address 239.255.255.253 and port 427. Since we are interested in pervasive environments, we consider that there is no directory agent. We then consider a subset of SLP message types:

- Service Request: UAs find service by type, scope, and search filter.
- Service Reply: SA returns Service URLs and their lifetimes.

- SAAdvert: SA sends its Service URL, scope, and attributes.

The SLP protocol formalization is illustrated in Figure 4.9.

```
SLP_GROUP_MANAGER    =  IDLE,
IDLE                 =  (join[slp] → MATCH
                     |  send[Multicast_groups] → IDLE
                     )  ,
MATCH                =
                     (  send[group : Multicast_groups] →
                          if (group == slp) then
                            (send[group] → MATCH
                            |leave → IDLE)
                          else
                            MATCH
                     |  leave → IDLE
                     )
SLP_RECEIVER         =  (join[slp] → LISTENING),
LISTENING            =  (send[slp] → LISTENING
                     |  send[slp] → serviceReply → LISTENING
                     |  send[slp].saadvert[i : DeviceRange] → serviceReply[i] → LISTENING
                     |  leave → STOP
                     )  +{join[Multicast_groups]}
//The join then advertise step
SLP_JOIN_ADVERTISE   =  (join[slp] → saadvert →STOP)/{saadvert/send[slp]}
//Discovery step
SLP_DISCOVERY        =  (serviceRequest →SLP_DISCOVERY
                     |  serviceReply[i : DeviceRange] →SLP_DISCOVERY
                     )  /{serviceRequest/send[slp]}
||SLP                =  (device[DeviceRange] :SLP_GROUP_MANAGER
                     || device[DeviceRange] :SLP_RECEIVER
                     || device[DeviceRange] :SLP_JOIN_ADVERTISE
                     || device[DeviceRange] :SLP_DISCOVERY
                     )
```

Figure 4.9: SLP specification

Achieving interoperability

Let us now apply the transparent interoperability specification in order to make SLP and SSDP interoperate. We first have to define the projection function that defines the semantics of the events (see Figure 4.11). Then, all the other processes: Switch, Switch′, W$_1$, W$_2$, M$_1$, M$_2$, Bridge$_1$ and Bridge$_2$ are automatically generated (see Figure 4.10).

To illustrate the functioning of the approach, consider an SLP client ($R1$) searching for a UPnP Service ($R2$). First, a $serviceRequest$ event is thrown, W$_1$ synchronizes with it and generates $a.tag.serviceRequest$. M$_1$ rises $a.tag.map.discover$ that is handled by Bridge$_1$ and transformed to $map.discover$, which makes Bridge$_2$ synchronize and throw $b.tag.map.discover$. Then, M$_2$ generates $b.tag.map.msearch$ that is transformed to $msearch$ by W$_1$. The UPnP device synchronizes with its glue and generates $response$ that synchronizes with W$_2$ and with UPNP_DISCOVERY (see Figure 4.8). W$_2$ generates $b.tag.response$ that is handled by M$_2$ and transformed to $b.tag.map.deviceDesc$ but none of the processes can synchronize with this event. On the other hand, UPNP_DISCOVERY throws an $httpget$ followed by $httpgetResponse$. The latter is handled by W$_2$ that generates $b.tag.httpgetResponse$ that is handled by M$_2$ and transformed to $b.tag.map.reply$. Then, Bridge$_2$ synchronizes and raises $map.reply$, which makes Bridge$_1$ throw $a.tag.map.reply$. Then, M$_2$ generates $a.tag.map.serviceReply$ that is transformed to $serviceReply$ by W$_1$. Then the SLP client receives the reply.

4.4.2 Example 2: Interoperability among different connector types

In this example, we go further by addressing interoperability between middleware based on different connector types: UPnP and Lime. We start by describing and formalizing each of them. Then, we discuss the applicability of the transparent interoperability approach (and in particular dynamic protocol translation) to this example.

$$
\begin{array}{lllll}
n & = & m & = & 1 \\
R1 & = & R1_1 & = & \text{SLP\_JOIN\_ADVERTISE}\|\text{SLP\_DISCOVERY}\|\text{SLP\_RECEIVER} \\
R2 & = & R2_1 & = & \text{UPNP\_JOIN\_ADVERTISE}\|\text{UPNP\_DISCOVERY}\|\text{UPNP\_RECEIVER} \\
 & & & & \|\text{UPNP\_SERVICE}\|\text{UPNP\_DESCRIPTION} \\
\text{Glue}_1 & & & = & \text{SLP\_GROUP\_MANAGER} \\
\text{Glue}_2 & = & \text{Glue}'_1 & = & \text{UPNP\_GROUP\_MANAGER} \\
E_1 & = & E1_1 & = & \{join[slp], saadvert, serviceRequest, serviceReply\} \\
\sum_{E1_1} & = & E1_1 & = & E_1 \\
E_2 & = & E2_1 & = & \{join[upnp], alive, msearch, response\} \\
\sum_{E2_1} & = & E2_1 & = & E_2 \\
I_1 & = & I1_1 & = & \{join[slp], saadvert, serviceRequest, serviceReply\} \\
O_1 & = & O1_1 & = & I_1 \\
\sum_{O1_1} & = & O1_1 & = & I_1 \\
I_2 & = & I2_1 & = & \{join[upnp], alive, msearch, response, httpget, httpgetResponse\} \\
O_2 & = & O2_1 & = & I_2 \\
\sum_{O2_1} & = & O2_1 & = & I_2
\end{array}
$$

//*Switch processes*

$$
\begin{array}{lll}
\text{Switch} & = & \text{Glue}_1 \\
\text{Switch}' & = & \text{Glue}_2
\end{array}
$$

//*Image protocol generation*

$$
\begin{array}{lllll}
\text{W}_1 & = & ToGlue_1 & = & join[slp] \rightarrow a.tag.join[slp] \rightarrow \text{W}_1 \;\mid\; a.tag.join[slp] \rightarrow join[slp] \rightarrow \text{W}_1 \\
 & & & \mid & saadvert \rightarrow a.tag.saadvert \rightarrow \text{W}_1 \;\mid\; a.tag.saadvert \rightarrow saadvert \rightarrow \text{W}_1 \\
 & & & \mid & serviceRequest \rightarrow a.tag.serviceRequest \rightarrow \text{W}_1 \;\mid\; a.tag.serviceRequest \rightarrow serviceRequest \rightarrow \text{W}_1 \\
 & & & \mid & serviceReply \rightarrow a.tag.serviceReply \rightarrow \text{W}_1 \;\mid\; a.tag.serviceReply \rightarrow serviceReply \rightarrow \text{W}_1 \\
 & & & \mid & a.reset \rightarrow \text{W}_1 \\
\text{W}_2 & = & ToGlue'_1 & = & join[upnp] \rightarrow b.tag.join[upnp] \rightarrow \text{W}_2 \;\mid\; b.tag.join[upnp] \rightarrow join[upnp] \rightarrow \text{W}_2 \\
 & & & \mid & alive \rightarrow b.tag.alive \rightarrow \text{W}_2 \;\mid\; b.tag.alive \rightarrow alive \rightarrow \text{W}_2 \\
 & & & \mid & msearch \rightarrow b.tag.msearch \rightarrow \text{W}_2 \;\mid\; b.tag.msearch \rightarrow msearch \rightarrow \text{W}_2 \\
 & & & \mid & response \rightarrow b.tag.response \rightarrow \text{W}_2 \;\mid\; b.tag.response \rightarrow response \rightarrow \text{W}_2 \\
 & & & \mid & httpget \rightarrow b.tag.httpget \rightarrow \text{W}_2 \;\mid\; b.tag.httpget \rightarrow httpget \rightarrow \text{W}_2 \\
 & & & \mid & httpgetResponse \rightarrow b.tag.httpgetResponse \rightarrow \text{W}_2 \;\mid\; b.tag.httpgetResponse \rightarrow httpgetResponse \rightarrow \text{W}_2 \\
 & & & \mid & b.reset \rightarrow \text{W}_2 \\
\text{M}_1 & = & ToMap_1 & = & a.tag.join[slp] \rightarrow a.tag.map.join \rightarrow \text{M}_1 \;\mid\; a.tag.map.join \rightarrow a.tag.join[slp] \rightarrow \text{M}_1 \\
 & & & \mid & a.tag.saadvert \rightarrow a.tag.map.advert \rightarrow \text{M}_1 \;\mid\; a.tag.map.advert \rightarrow a.tag.saadvert \rightarrow \text{M}_1 \\
 & & & \mid & a.tag.serviceRequest \rightarrow a.tag.map.discover \rightarrow \text{M}_1 \;\mid\; a.tag.map.discover \rightarrow a.tag.serviceRequest \rightarrow \text{M}_1 \\
 & & & \mid & a.tag.serviceReply \rightarrow a.tag.map.reply \rightarrow \text{M}_1 \;\mid\; a.tag.map.reply \rightarrow a.tag.serviceReply \rightarrow \text{M}_1 \\
 & & & \mid & a.reset \rightarrow \text{M}_1 \\
\text{M}_2 & = & ToMap'_1 & = & b.tag.join[upnp] \rightarrow b.tag.map.join \rightarrow \text{M}_2 \;\mid\; b.tag.map.join \rightarrow b.tag.join[upnp] \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.join[upnp] \rightarrow b.tag.map.join \rightarrow \text{M}_2 \;\mid\; b.tag.map.join \rightarrow b.tag.join[upnp] \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.alive \rightarrow b.tag.map.advert \rightarrow \text{M}_2 \;\mid\; b.tag.map.advert \rightarrow b.tag.alive \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.msearch \rightarrow b.tag.map.discover \rightarrow \text{M}_2 \;\mid\; b.tag.map.discover \rightarrow b.tag.msearch \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.response \rightarrow b.tag.map.deviceDesc \rightarrow \text{M}_2 \;\mid\; b.tag.map.deviceDesc \rightarrow b.tag.response \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.httpget \rightarrow b.tag.map.serviceDscv \rightarrow \text{M}_2 \;\mid\; b.tag.map.serviceDscv \rightarrow b.tag.httpget \rightarrow \text{M}_2 \\
 & & & \mid & b.tag.httpgetResponse \rightarrow b.tag.map.reply \rightarrow \text{M}_2 \;\mid\; b.tag.map.reply \rightarrow b.tag.httpgetResponse \rightarrow \text{M}_2 \\
 & & & \mid & b.reset \rightarrow \text{M}_2
\end{array}
$$

//*Bridging*

$$
\begin{array}{lllll}
\text{Bridge}_1 & = & ToBridge_1 & = & a.tag.map.join \rightarrow map.join \rightarrow \text{Bridge}_1 \\
 & & & \mid & a.tag.map.advert \rightarrow map.advert \rightarrow \text{Bridge}_1 \\
 & & & \mid & a.tag.map.discover \rightarrow map.discover \rightarrow \text{Bridge}_1 \\
 & & & \mid & a.tag.map.reply \rightarrow map.reply \rightarrow \text{Bridge}_1 \\
 & & & \mid & map.join \rightarrow a.tag.map.join \rightarrow \text{Bridge}_1 \\
 & & & \mid & map.advert \rightarrow a.tag.map.advert \rightarrow \text{Bridge}_1 \\
 & & & \mid & map.discover \rightarrow a.tag.map.discover \rightarrow \text{Bridge}_1 \\
 & & & \mid & map.reply \rightarrow a.tag.map.reply \rightarrow \text{Bridge}_1 \\
 & & & \mid & a.reset \rightarrow \text{Bridge}_1 \\
\text{Bridge}_2 & = & ToBridge'_1 & = & b.tag.map.join \rightarrow map.join \rightarrow \text{Bridge}_2 \\
 & & & \mid & b.tag.map.advert \rightarrow map.advert \rightarrow \text{Bridge}_2 \\
 & & & \mid & b.tag.map.discover \rightarrow map.discover \rightarrow \text{Bridge}_2 \\
 & & & \mid & b.tag.map.reply \rightarrow map.reply \rightarrow \text{Bridge}_2 \\
 & & & \mid & map.join \rightarrow b.tag.map.join \rightarrow \text{Bridge}_2 \\
 & & & \mid & map.advert \rightarrow b.tag.map.advert \rightarrow \text{Bridge}_2 \\
 & & & \mid & map.discover \rightarrow b.tag.map.discover \rightarrow \text{Bridge}_2 \\
 & & & \mid & map.reply \rightarrow b.tag.map.reply \rightarrow \text{Bridge}_2 \\
 & & & \mid & b.reset \rightarrow \text{Bridge}_2
\end{array}
$$

//*The* `Connector`

$$
\begin{array}{lll}
\|\text{C-Transparent\_Interop} & = & R1\|\ \text{W}\ _1\|\ \text{M}\ _1 \\
 & & \|\text{Glue}_1/\{f(r:\alpha\text{Glue}_1)/[r]\}\|\text{Bridge}_1 \\
 & & \|\text{Bridge}_2\|\ \text{Glue}_2/\{f(r:\alpha\text{Glue}_2)/[r]\} \\
 & & \|\text{M}\ _2\|\ \text{W}\ _2\|R2
\end{array}
$$

Figure 4.10: Application of the transparent interoperability approach to SLP-SSDP

$$
\begin{array}{lcl|lcl}
f(join[slp]) & = & map.join & f(join[upnp]) & = & map.join \\
f(saadvert) & = & map.advert & f(alive) & = & map.advert \\
f(serviceRequest) & = & map.discover & f(msearch) & = & map.discover \\
f(serviceReply) & = & map.reply & f(response) & = & map.deviceDesc \\
 & & & f(httpget) & = & map.serviceDscv \\
 & & & f(httpgetResponse) & = & map.reply
\end{array}
$$

Figure 4.11: Projection function

Universal Plug and Play (UPnP)

A UPnP device can be any entity on the network that implements the protocols required by the UPnP architecture [37]. Each UPnP device implements zero or more services. A service represents a functionality provided by the device. Each service has a set of actions that represents the methods offered by the service. A control point is an entity on the network that asks for a functionality provided by a device. In other words, the control point behaves as a client invoking actions on services provided by the device. UPnP defines the following phases:

- **Addressing.** The device or the control point joins the network.
- **Advertising.** The device multicasts its device description.
- **Discovery.** The control point searches for a device. The device replies by sending its device description.
- **Description.** Once the control point gets the device description, it addresses a request to ask for the description of one of its services.
- **Control.** Once the control point gets the service description, it invokes one of the service actions.

UPnP relies on SSDP (see Section 4.4.1) to get the service description and then on SOAP to invoke its method. Figure 4.12 presents the FSP-based formalization of UPnP.

$$
\begin{array}{lcl}
\text{ACTION} & = & (soapRequest \rightarrow compute \rightarrow soapResponse \rightarrow \text{ACTION} \\
\text{CONTROL} & = & (control \rightarrow soapRequest \rightarrow soapResponse \rightarrow \text{CONTROL} \\
\|\text{A\_UPNP\_MW} & = & (\text{SSDP} \\
 & \| & service[i : DeviceRange][j : ServiceRangei][k : ActionRangeij] \text{ :ACTION} \\
 & \| & service[i : DeviceRange][j : ServiceRangei][k : ActionRangeij] \text{ :CONTROL} \\
 &) & /\{service[i : DeviceRange][j : ServiceRangei].control/ \\
 & & service[i][j][ActionRangeij].control\}
\end{array}
$$

Figure 4.12: UPnP specification

Linda in a Mobile environment (Lime)

Lime is a Java-based middleware that provides a coordination layer that can be exploited for designing applications that exhibit either logical or physical mobility [57]. In Linda, processes communicate by writing, reading, and removing data from a tuple space that is assumed to be persistent and globally shared among all processes. Lime adapts this notion to a mobile environment by breaking down the notion of a global tuple space, and distributing its contents across multiple mobile components. Lime also introduces the notions of tuple location, for querying a partition of the federated tuple space, and of reactive programming, to allow actions to be performed with varying degrees of atomicity upon insertion of a tuple. Tuple spaces are also extended with a notion of location and programs are given the ability to react to specified states. Lime explicitly extends the basic Linda tuple space with the notion of reaction. The formal specification of the Lime middleware is illustrated in Figure 4.13.

```
set Tuples                    =   {tuple1, tuple2}
range TupleUsersRange         =   1..N
//Linda specification
TUPLE(T = 'any)               =   TUPLE[0];
TUPLE[i : 0..N]               =   (out[T]                        →
                                     if (i < N) then
                                        TUPLE[i + 1]
                              |   when (i > 0)in[T]              →   TUPLE[i − 1]
                              |   when (i > 0)inp[True][T]       →   TUPLE[i − 1]
                              |   when (i == 0)inp[False][T]     →   TUPLE[i]
                              |   when (i > 0)rdg[T]             →   TUPLE[i]
                              |   rdp[i > 0][T]                  →   TUPLE[i]
                              )
//Reaction specification
NOTIFICATION_MANAGER          =   IDLE,
IDLE                          =   (reactsTo[patt : Tuples]       →   MATCH[patt]
                              |   out[Tuples]                    →   IDLE
                              ),
MATCH[patt : Tuples]          =   (out[tuple : Tuples]           →
                                     if (tuple == patt) then
                                        (notification[tuple] → MATCH[patt]
                                        |leave → IDLE)
                                     else
                                        MATCH[patt]
                              |   leave                          →   IDLE
                              )
||NOTIFICATIONS_MANAGER       =   (user[TupleUsersRange] :NOTIFICATION_MANAGER)
                                  /{out/user[TupleUsersRange].out}
SENDER                        =   (out[Tuples]                   →   SENDER)
RECEIVER(P = 'pattern)        =   (reactsTo[P]                   →   LISTENING),
LISTENING                     =   (notification[P]               →   LISTENING
                              |   notification[P]                →   leave →STOP
                              ) + {join[Tuples]}
||LIME                        =   ((forall [t : Tuples] TUPLE(t))
                              ||  user[TupleUsersRange] :RECEIVER('tuple1)
                              ||  user[TupleUsersRange] :RECEIVER('tuple2)
                              ||  user[TupleUsersRange] :RECEIVER('tuple3)
                              ||  NOTIFICATIONS_MANAGER
                              ||  SENDER)
```

Figure 4.13: Lime specification

Achieving interoperability

Connectors of the same type adhere to the same abstract protocol, which is the image protocol. However, when considering different types of protocols it is harder, if not impossible, to find such an abstract protocol since the coordination paradigms are divergent even if they both aim at making the components communicate. As a consequence it is not possible to find the set of semantic events (represented by the projection function) shared between the connectors because there is a big difference in the semantics of different types of connectors. In this case, the procedure call connector of UPnP uses two basic primitives, *Send* and *Receive*, while the data access connector of Lime provides shared address space through *Read* and *Write* primitives. Thus, a new approach should be defined to address this issue.

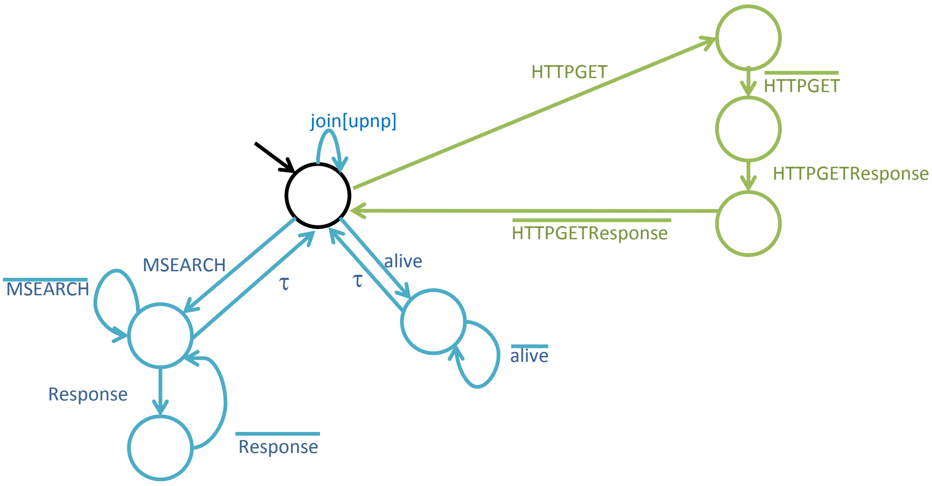

HTTPGET
$\overline{\text{HTTPGET}}$
join[upnp]
HTTPGETResponse
$\overline{\text{HTTPGETResponse}}$
MSEARCH
$\overline{\text{MSEARCH}}$
τ
τ
alive
$\overline{\text{alive}}$
Response
$\overline{\text{Response}}$

Figure 4.14: LTS of the SSDP glue

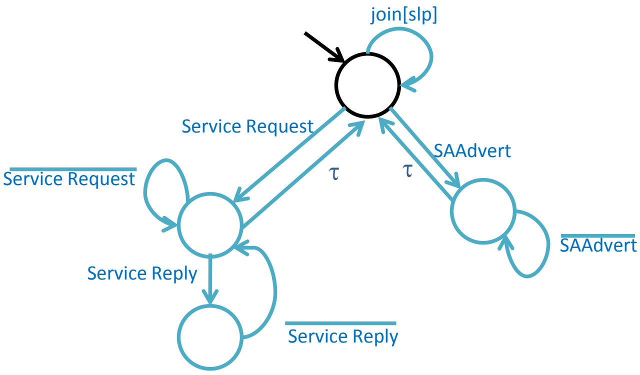
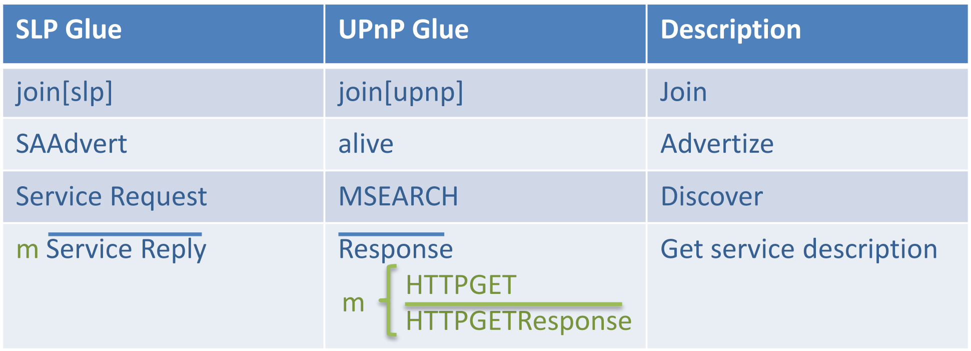

join[slp]
Service Request
$\overline{\text{Service Request}}$
τ
τ
SAAdvert
$\overline{\text{SAAdvert}}$
Service Reply
$\overline{\text{Service Reply}}$

Figure 4.15: LTS of the SLP glue

4.5 Middleware-layer Interoperability versus Application-layer Interoperability

As an alternative approach to middleware interoperability, we study the applicability to the middleware-layer of our approach to application-layer interoperability approach introduced in Chapter 3. To this end, we use the same examples as in Section 4.4.



SLP Glue	UPnP Glue	Description
join[slp]	join[upnp]	Join
SAAdvert	alive	Advertize
Service Request	MSEARCH	Discover
m $\overline{\text{Service Reply}}$	$\overline{\text{Response}}$ m { HTTPGET $\overline{\text{HTTPGETResponse}}$	Get service description

Figure 4.16: SLP/SSDP ontology mapping

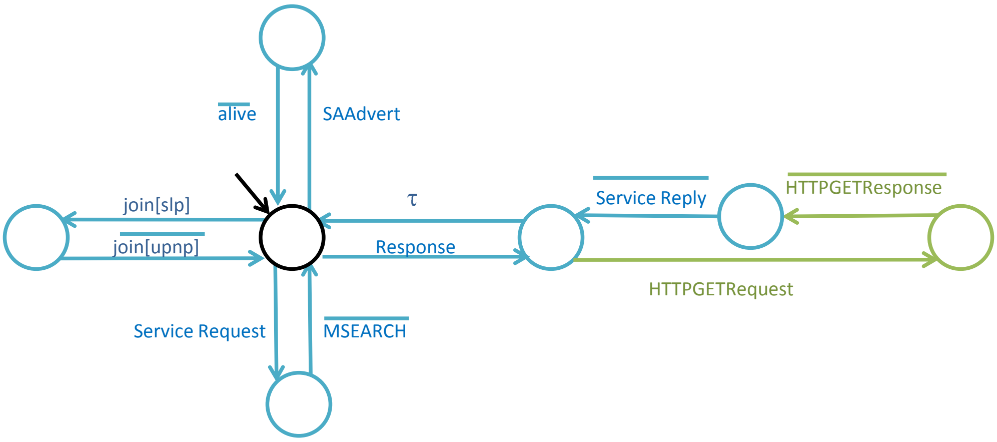

$\overline{\text{alive}}$
SAAdvert
join[slp]
$\overline{\text{join[upnp]}}$
τ
Response
$\overline{\text{Service Reply}}$
$\overline{\text{HTTPGETResponse}}$
HTTPGETRequest
Service Request
$\overline{\text{MSEARCH}}$

Figure 4.17: LTS of the SLP/SSDP mediator

4.5.1 Example 1: Interoperability within the same connector type

We recall that in this example, we would like to adapt the SLP protocol to the SSDP protocol using the application-layer interoperability approach. This approach relies on the LTSs of the protocols to be made interoperable and on the ontology mapping of their primitives. The required LTS are generated from the FSP models described in Section 4.4. However, they are abstracted due to state explosion. Figures 4.14 and 4.15 illustrate the abstraction of the SSDP glue and the SLP glue respectively. Their ontology mapping is represented in Figure 4.16.

By applying the approach, we obtain the mediator illustrated in Figure 4.17.

4.5.2 Example 2: Interoperability among different connector types

We recall that in this example, our target is to make UPnP interoperate with Lime using the application-layer interoperability approach. Figure 4.18 and 4.19 illustrate the abstraction of the UPnP glue and Lime glue respectively. However, it is not possible, to the best of our knowledge, to establish an ontology mapping between them.

As for the transparent interoperability approach, when considering connectors of the same type, we can define an ontology/event mapping since both of them use the same abstract model associated with the interaction service that is realized. However, when considering connectors of different types, it is hard to define a direct ontology/event mapping between their primitives.

Moreover, structural matching that is applied at the application-layer is too constraining for middleware-layer interoperability, particularly, when addressing components running middleware based on different coordination models. On the other hand, at the middleware layer, we may consider the protocols known in

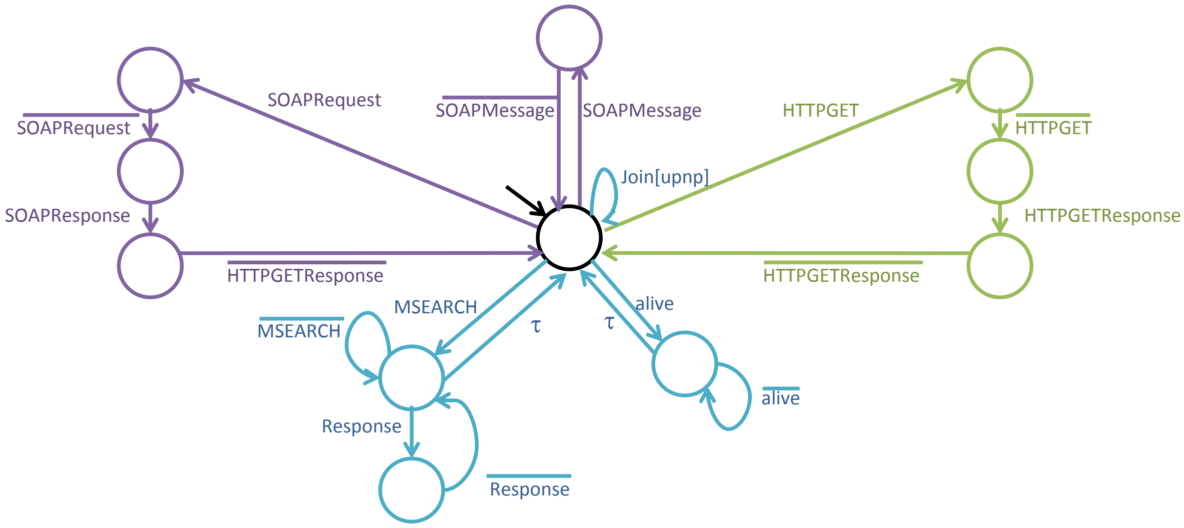

$\overline{\text{SOAPRequest}}$
SOAPRequest
$\overline{\text{SOAPMessage}}$
SOAPMessage
HTTPGET
$\overline{\text{HTTPGET}}$
Join[upnp]
SOAPResponse
HTTPGETResponse
$\overline{\text{HTTPGETResponse}}$
$\overline{\text{HTTPGETResponse}}$
MSEARCH
$\overline{\text{MSEARCH}}$
τ
τ
alive
$\overline{\text{alive}}$
Response
$\overline{\text{Response}}$

Figure 4.18: LTS of the UPnP glue

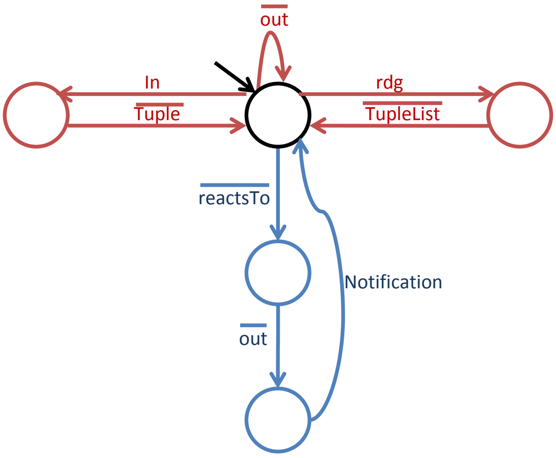

$\overline{\text{out}}$
In
$\overline{\text{Tuple}}$
rdg
$\overline{\text{TupleList}}$
$\overline{\text{reactsTo}}$
Notification
$\overline{\text{out}}$

Figure 4.19: LTS of the Lime glue

advance, which is not possible at the application layer. We then get another perspective on interoperability.

4.6 Summary

In this chapter, we first established the relation between connectors and middleware. We used an FSP-based formalization to specify the existing approaches to middleware interoperability, which are informally discussed in [2]. We assessed the transparent interoperability approach (the one based on dynamic protocol synthesis, as aimed by CONNECT) through two different examples, the first one addresses heterogeneity between connectors of the same type while the second tackles heterogeneity between connectors of different types. The same two examples were also used to evaluate the application-layer interoperability approach, which was described in Chapter 3, for middleware-layer protocols.

Both approaches succeed to manage interoperability among connectors of the same type but fail to achieve interoperability among connectors of different types. Indeed, both of them require the communicating parties to use primitives that have the same semantics. This is expressed in the transparent interoperability approach by the use of the projection function, and for application-layer interoperability by the existence of an ontology mapping.

These approaches also focus on the control flow between the communicating parties. However, the exchanged data and its semantics, might be a valuable dimension that has to be considered. Existing approaches to middleware interoperability have to be enhanced in order to tackle broader heterogeneity and to handle not only connectors of the same type but also of different types. This can only be achieved by finding the right abstraction that captures the similarities of different connectors regardless of the type they belong to.

5 Application-layer Protocol Elicitation: Towards an Automated Model-based Approach

As discussed in [1], one of the key challenges of CONNECT concerns the possibility to characterize Networked Systems (NSs) semantically. This means that besides a syntactical description of the NS signature, e.g., by means of either the WSDL notation [6] or an IDL (*Interface Definition Language*) description, there is the need in the CONNECTor synthesis process for pieces of semantic information about the functionality the NS provides. As discussed in Chapter 3, these pieces of information can describe different views of the system semantics, from ontological ones that ease discovery, to behavioral ones that ease synthesis. For the latter many approaches have been proposed in the literature with the aim to automatically synthesize composition/coordination/mediation code for a set of heterogeneous NSs, see [11, 17, 21, 22, 35, 36, 40, 46, 47, 50, 54, 56, 60, 64, 73, 76] just to mention the most recent. These approaches rely on the assumption that, along with the syntactical description of the NS signature, some information is provided about how other systems interacting with the NS should behave. We call this behavioral information the system *behavior protocol*. Unfortunately, in the application scenarios envisioned by CONNECT [1], this assumption turns out to be unfounded.

This is the problem we address in this chapter: *how to compensate for the lack of information about a NS's behavioral assumptions?*. Note that this problem is related to behavior protocol learning issues investigated in WP4 [3]. However, the work described in this chapter has to be considered as complementary, and not as substitutive, of CONNECT learning algorithms. Actually, as discussed in detail in Section 5.5, the work in this chapter presents several differences with respect to work in WP4. It should be considered as work developed within WP3 in order to understand the requirements, for the work in WP4, that enable automated CONNECTor synthesis. During the first year of the project, the work described in this chapter allowed, on one side, WP3 to work in parallel with WP4, and on the other side, for the definition of the relationships among the two WPs. Therefore, the concepts underlying the work described in this chapter can be seen as a bridge between the CONNECT synthesis process and the learning algorithms. Furthermore, note also that protocol elicitation does make sense for application-layer protocols only. Middleware-layer protocols have to be part of either the interface or some given knowledge base.

In this chapter, we present the work published in [14], which has been applied to the context of Web services (WSs). Thus, the work in this chapter considers only WSs as possible NSs. Note that this is not a limitation since the theoretical core of the approach is general enough to be applied also to other application contexts. In fact, considering WSs just requires to start from a WSDL description instead of starting from another type of syntactical signature description, e.g., Microsoft IDL, Java IDL, OMG IDL. The approach we present, called `StrawBerry` (Synthesized Tested Refined Automaton of Web service BEhavior pRotocol), is a method for the automatic discovery of the behavior protocol of a WS. Since for a published WS, in practice, only its *signature syntactical description*, i.e., its WSDL, can generally be assumed to be available, `StrawBerry` derives from the WSDL, in an automated way, a partial ordering relation among the invocations of the different WSDL operations, which we represent as an automaton. This automaton, called *Behavior Protocol automaton*, models the interaction protocol that a client has to abide by to correctly interact with the WS. This automaton also explicitly models the data that has to be passed to the WS operations. More precisely, the states of the behavior protocol automaton are WS execution states and the transitions, labelled with operation names plus I/O data, model possible operation invocations from the client of the WS.

The behavior protocol is obtained through synthesis and testing stages. The synthesis stage is driven by data type analysis, through which we obtain a preliminary dependencies automaton, that can be optimized by means of heuristics. Once synthesized, this dependencies automaton is validated through testing against the WS's implementation to verify conformance, and finally transformed into the behavior protocol.

`StrawBerry` is a black-box and extra-procedural method. It is black-box since it takes into account only the WSDL of the WS. It is extra-procedural since it focuses on synthesizing a model of the behavior that is assumed when interacting with the WS from outside, as opposed to intra-procedural methods that synthesize a model of the implementation logic of the single WS operations [44, 82, 83]. In fact, the behavior protocol obtained through `StrawBerry` enables the automated orchestration of WSs.

This chapter is organized as follows. Section 5.1 presents the actual technological scenario in which

StrawBerry works and discusses the underlying programming assumptions. In Section 5.2, by means of an explanatory example, we introduce the StrawBerry method. Section 5.3 presents the method formalization. In Section 5.4, we show an application of StrawBerry to a WS existing on the Web, i.e., the *Amazon E-Commerce Service*. In Section 5.5, we relate StrawBerry to other similar approaches discussing also differences with work developed in WP4. In Section 5.6, we give some concluding remarks on the presented approach.

5.1 Setting the Context

A WS is typically constructed over HTTP and it is by default a *state-less* software entity. That is, no state internal to the WS exists during a *complex* interaction with the a client application. This is not the best solution for many application scenarios, e.g., e-commerce. Some technologies have been proposed to allow the development of *state-full* WSs through the implementation of the concept of *session*. Informally, a session consists of a set of attributes (set of data with name, type, and value) that characterize an invoked sequence of WS operations. Typically, a session is realized to be persistent during a complete WS interaction with the client. Different approaches have been proposed to realize a session: (i) by using the well-known mechanism of *cookies*; (ii) by using WS-oriented APIs[1], or, at a lower level, by means of session IDs associated to the header of the SOAP messages; (iii) based on the WS-ReliableMessaging standard[2]; or (iv) by ad-hoc programming, that mixes data used for managing the session with business logic data[3].

Each of these approaches has its own advantages and disadvantages. Techniques (i) and (ii) keep the business logic separated from the logic used to manage the session. However, in order to use these techniques, the client code must be aware of the session WS capabilities. Technique (iii) also keeps the business logic separated from the session management logic. Furthermore, session management is completely transparent to the client since it is demanded to a framework on top of which the WS is built. However the client application must support the particular implementation of the framework. Using technique (iv), WSs keep their *state-less* nature, and a session is implicitly realized by passing the relative data (i.e., data encoding the WS state) from one operation to another. Therefore, session data are explicitly added as I/O data of the WS operations. The disadvantage of this technique is that the data concerning both business and session logic are mixed in the WSDL. However the client application needs only to rely on the WSDL interface in order to interact with the WS. This promotes WS reuse and interoperability among different WSs.

It is worthwhile noticing that BPEL (*Business Process Execution Language*) orchestration (which means standard WS composition/coordination/mediation) cannot be realized with WSs developed by using techniques (i), (ii), and (iii). A BPEL connector cannot use such (hidden) techniques to enable a session management. Since the present standard for WS composition is BPEL and an important number of relevant WSs, like Amazon, follows technique (iv), this is also the programming assumption for StrawBerry.

5.2 Method Description

In this section we provide an overview of the StrawBerry method (Section 5.2.1). Then, by means of a simple explanatory example (Section 5.2.2), we informally introduce its steps (Section 5.2.3).

5.2.1 Overview

StrawBerry takes as input the WSDL of a WS, and returns an automaton modeling its behavior protocol (client side). Figure 5.1 graphically represents StrawBerry as a process split into five main activities. The *Dependencies Elicitation* activity elicits data dependencies between the I/O parameters of the operations defined in the WSDL. A dependency is recorded whenever the type of the output of an operation matches

[1] e.g., JAX-WS: http://weblogs.java.net/blog/ramapulavarthi/archive/2006/06/ maintaining_ses.html

[2] WS-ReliableMessaging standard: http://weblogs.java.net/blog/mikeg/ archive/2006/08/wsreliable_mess.html

[3] As it is done, e.g., for the *Amazon* e-commerce service: http:// webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl.

with the type of the input of another operation. The match is syntactic. The elicited set of I/O dependencies may be optimized under some heuristics concerning the syntactic characteristics of the WSDL. The elicited set of I/O dependencies (see the *Input/Output Dependencies* artifact shown in Figure 5.1) is used for constructing a data-flow model (see the *Saturated Dependencies Automaton Synthesis* activity and the *Saturated Dependencies Automaton* artifact shown in Figure 5.1) where each node stores data dependencies that concern the output parameters of a specific operation and directed arcs are used to model syntactic matches between output parameters of an operation and input parameters of another operation. This model is completed by applying a *saturation rule*. This rule adds new dependencies that model the possibility for the client to invoke a WS operation by directly providing its input parameters. The resulting automaton is then validated against the implementation of the WS through testing (see *Dependencies Automaton Refinement Through Testing* activity shown in Figure 5.1).

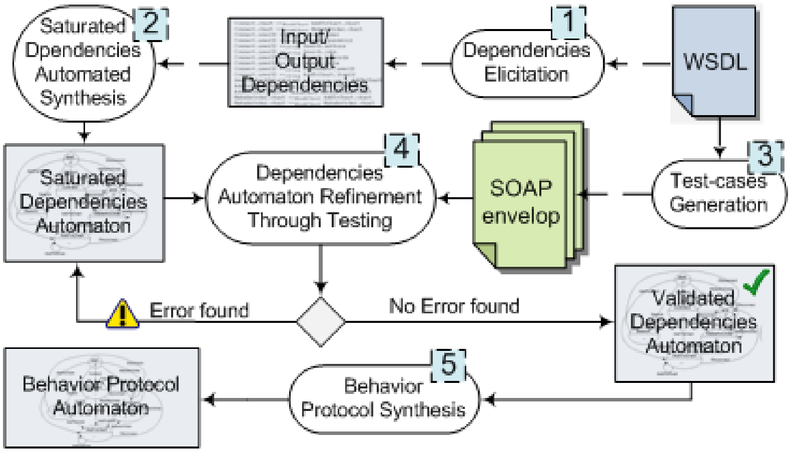

Saturated Dpendencies Automated Synthesis
2
Input/ Output Dependencies
Dependencies Elicitation
1
WSDL
Saturated Dependencies Automaton
Dependencies Automaton Refinement Through Testing
4
SOAP envelop
Test-cases Generation
3
Error found
No Error found
Validated Dependencies Automaton
Behavior Protocol Automaton
Behavior Protocol Synthesis
5

Figure 5.1: Overview of the StrawBerry method

The testing phase takes as input the SOAP messages produced by the *Test-cases generation* activity. The latter, driven by coverage criteria, automatically derives a suite of test cases (i.e., SOAP envelope messages). For this purpose, we use the WS-TAXI [10] tool that takes as input the WSDL of a WS and automatically produces the SOAP envelope messages ready for execution. Note that testing is used here in opposite way with respect to the usual model-based testing (MBT) practice [78]. In fact, in MBT the automaton is used as an oracle, and testing aims at checking whether the implementation conforms to it. In StrawBerry instead, the oracle is based on the implementation and testing aims at validating whether the synthesized automaton is a correct data-flow abstraction of it. Intuitively, we can say that StrawBerry tests if the model conforms to the implementation. Testing is used to refine the syntactic dependencies by discovering those that are semantically wrong. By construction, the inferred set of dependencies is syntactically correct. However, it might not be correct semantically since it may contain false positives. If during the testing phase an error is found, this means that the automaton must be refined since the set of I/O dependencies contains false dependencies.

Once the testing phase is successfully terminated, the final automaton models, following a data-flow paradigm, the set of validated "chains" of data dependencies. StrawBerry terminates by transforming this data-flow model into a control-flow model (see the *Behavior Protocol Synthesis* activity in Figure 5.1). This is another kind of automaton whose nodes are WS execution states and whose transitions, labelled with operation names plus I/O data, model the possible operation invocations from the client to the WS.

5.2.2 Explanatory example

The explanatory example that we use in this chapter is a WS of an online bookshop that we call WS_Lib. This WS defines, in its WSDL, the following operations:

- Connect: in order to login, a registered user inserts his/her username and password. This operation returns an empty cart and a userID (to uniquely identify the session). If something wrong occurs (e.g., wrong `user` or `password`), an error message is returned; an output parameter of name `errmsg` is also included for all the other operations.

Input data	Output data
`user: string;`	`cart: BookCart;`
`password: string;`	`userID: string;`
	`errmsg: string;`

• `Disconnect`: it is used by a logged user to log out and close the session.

Input data	Output data
`userID: string;`	`regularResponse: string;`
	`errmsg: string;`

• `Search`: it is used to search the bookshop catalogue by means of different search criteria, namely, by authors, ISBN, keywords, and title, and returns a list of books satisfying the search criteria.

Input data	Output data
`authors: string;`	`bookDetailsList: BookDetailsList;`
`isbn: string;`	`errmsg: string;`
`keywords: string;`	
`title: string;`	

• `AddToCart`: it adds a book from a list to the cart associated to the user.

Input data	Output data
`itemId: string;`	`cart: BookCart;`
`itemList: BookDetailsList;`	`errmsg: string;`
`cart: BookCart;`	

• `MakeAnOrder`: it makes an order of the books contained in the cart. When the order has been made the cart is emptied.

Input data	Output data
`cart: BookCart;`	`cart: BookCart;`
	`errmsg: string;`

Data that characterize a WS session are: `userID` (user identifier), `cart` (the cart associated to the user), `bookDetailsList` and `itemList` (the list of books that match the search criteria).

5.2.3 Stepwise description

By referring to Figure 5.1, we show an overview of how our approach would process the `WS_Lib` WSDL.

Activity 1: Dependencies elicitation.
This activity is split into the following two steps. The first step is mandatory and it is the true dependencies elicitation step. The second is optional and performs an optimization through some heuristics.

Step 1.1, dependency set elicitation: `StrawBerry` automatically derives a "flat" version of the WSDL. This flattening process aims at making the structure of the I/O messages of the WSDL operations explicit with respect to the element types defined in the XML schema of the WSDL. Starting from the flattened WSDL, by syntactically matching the type of an output element of an operation with the type of an input element of another operation, `StrawBerry` automatically elicits the following set of data dependencies:

`Connect.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`Connect.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`
`Connect.userID` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`Connect.errmsg` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`Disconnect.regularResponse` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`Disconnect.errmsg` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`Search.bookDetailsList` $\mapsto_{BookDetailsList}$ `AddToCart.itemList`
`Search.errmsg` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`AddToCart.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`AddToCart.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`
`AddToCart.errmsg` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$
`MakeAnOrder.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`MakeAnOrder.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`
`MakeAnOrder.errmsg` $\mapsto_{string}$ `x`, for each `x`$\in I_{string}$

where: I_{string}=\{`Connect.user`,`Connect.password`,`Search.authors`, `Search.isbn`,`Search.keywords`, `Search.title`,`Disconnect.userID`, `AddToCart.itemId`\}.

For instance, `Search.bookDetailsList` $\mapsto_{BookDetailsList}$ `AddToCart.itemList` means that, the value of `bookDetailsList`, as output of `Search`, can be set as input parameter `itemList` of `AddToCart`.

Given a data dependency, we refer to its left hand-side operation as the *source* operation, and to the right hand-side operation as the *sink* operation. Dependencies are labelled as *certain* or *uncertain*. Initially all derived dependencies are marked as uncertain; as we collect more evidence (which happens via testing or through application of heuristics), uncertain dependencies are either eliminated or promoted to certain.

Step 1.2, dependency set optimization: this optimization step can be enabled/disabled by the `StrawBerry` user. It aims at identifying those dependencies that can be already removed or considered as certain, hence preventing waste useless test resources in the fourth activity of our method. This step is currently based on the following three heuristics (as we accumulate further experience, smarter heuristics can be introduced).

• *Heuristic 1*: all dependencies defined on a "complex type" are considered as certain. This heuristic comes out from the observation that a dependency defined on a complex type is certain with a high probability due to the specificity of that type. In our approach, as complex type, we consider XML Schema types defined by means of the `complexType` and `simpleType` tags (e.g., sequence, choice, all, restriction and extension of a base type).

• *Heuristic 2*: all dependencies defined between data parameters having the same name (and the same type) are considered as certain. This heuristic comes out from usual programming practice.

• *Heuristic 3*: all dependencies defined between an output parameter that is interpreted as an error and an input parameter can be removed. In our example, `errmsg` is a string. Error outputs should be never matched since they represent the end of an interaction. Thus, if the `StrawBerry`'s user has this information this heuristic can be enabled to prune the set of dependencies.

Coming back to our explanatory example, after the application of *Heuristic 1*, the following dependencies are considered as certain:

`Connect.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`Connect.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`
`Search.bookDetailsList` $\mapsto_{BookDetailsList}$ `AddToCart.itemList`
`AddToCart.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`AddToCart.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`
`MakeAnOrder.cart` $\mapsto_{BookCart}$ `AddToCart.cart`
`MakeAnOrder.cart` $\mapsto_{BookCart}$ `MakeAnOrder.cart`

Note that in this case it is possible to perform a `MakeAnOrder` operation after another `MakeAnOrder` even though the `cart` is empty (the same hold for a `MakeAnOrder` operation after `Connect`) since the WS implementation considers this behavior as correct. In another scenario this could be considered as an error thus preventing the use of *Heuristic 1*.

Now, if we apply *Heuristic 2*, another dependency is considered as certain: `Connect.userID` $\mapsto_{string}$ `Disconnect.userID`. Finally, if we apply *Heuristic 3*, the following dependencies are removed:

Connect.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
Disconnect.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
Search.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
AddToCart.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
MakeAnOrder.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$

Activity 2: Saturated dependencies automaton synthesis.

Step 2.1, node generation: once the data dependencies are elicited, StrawBerry synthesizes the dependencies automaton. To do this StrawBerry builds a node for each WSDL operation that has at least one elicited dependency.

In Figure 5.2, we show the nodes built for WS_Lib. A node stores the name of the operation and the data dependencies defined on its output parameters. In Figure 5.2, *certain* dependencies are identified by the tick (✓).

Step 2.2, Dependencies automaton synthesis: each arc from a node to another node reflects the data dependencies stored in the source node. The dependencies automaton for WS_Lib is shown in Figure 5.3. The Env node and the dotted lines represent the node and the arcs added by the saturation phase explained in the following step.

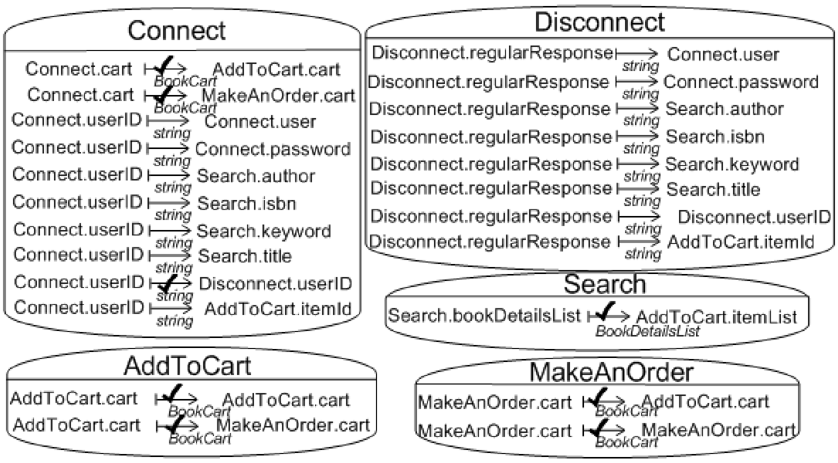

Connect
Connect.cart ✓⟼ AddToCart.cart BookCart
Connect.cart ✓⟼ MakeAnOrder.cart BookCart
Connect.userID ⟼ Connect.user string
Connect.userID ⟼ Connect.password string
Connect.userID ⟼ Search.author string
Connect.userID ⟼ Search.isbn string
Connect.userID ⟼ Search.keyword string
Connect.userID ⟼ Search.title string
Connect.userID ✓⟼ Disconnect.userID string
Connect.userID ⟼ AddToCart.itemId string
Disconnect
Disconnect.regularResponse ⟼ Connect.user string
Disconnect.regularResponse ⟼ Connect.password string
Disconnect.regularResponse ⟼ Search.author string
Disconnect.regularResponse ⟼ Search.isbn string
Disconnect.regularResponse ⟼ Search.keyword string
Disconnect.regularResponse ⟼ Search.title string
Disconnect.regularResponse ⟼ Disconnect.userID string
Disconnect.regularResponse ⟼ AddToCart.itemId string
Search
Search.bookDetailsList ✓⟼ AddToCart.itemList BookDetailsList
AddToCart
AddToCart.cart ✓⟼ AddToCart.cart BookCart
AddToCart.cart ✓⟼ MakeAnOrder.cart BookCart
MakeAnOrder
MakeAnOrder.cart ✓⟼ AddToCart.cart BookCart
MakeAnOrder.cart ✓⟼ MakeAnOrder.cart BookCart

Figure 5.2: Generated nodes

Step 2.3, Dependencies automaton saturation: for testing purposes we need to take into account also the possibility for the client to directly provide inputs to the WS operations. Thus, we add a new node, Env. This node stores *uncertain* dependencies conforming to the pattern: $\bigstar\mapsto_T$Op.p for each sink operation Op and for each input parameter p of Op of type T. The symbol $\bigstar$ denotes a datum directly provided by the client. For the sake of presentation, we do not show the content stored into Env. According to the dependencies stored into Env, the saturation step adds an arc from Env to every other depending node.

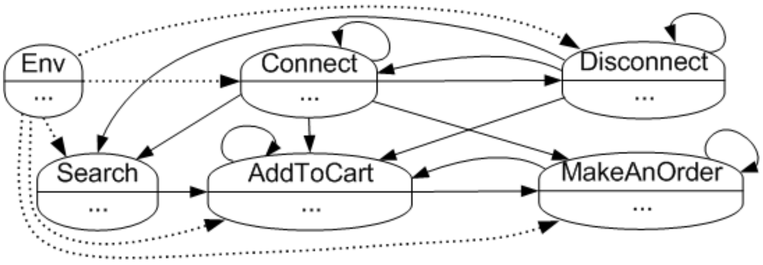


Figure 5.3: Saturated dependencies automaton

Activity 3: Test-cases generation.

As said, the only input to StrawBerry is a WSDL description. From it, StrawBerry derives the black-box test cases that are used in the testing stage (see next activity). Since the test subject is a WS, a test case consists of a SOAP envelope message whose input parameters are filled with appropriate data values. There exist several tools that help to automatically derive such test cases from WSDL, among which soapUI[4]. is probably the most popular. StrawBerry adopts the WS-TAXI tool [10] that enhances soapUI by fully automating test case derivation. Since it is not crucial for the purposes of the work described in this chapter, we do not provide all the details of the WS-TAXI functioning which can be found in [10]. It is worth however to clarify how WS-TAXI deals with input parameter values.

Listing 5.1 shows an example of a SOAP envelope message produced by WS-TAXI for testing Search. This test case aims at performing a book search based on the authors criterion. In Listing 5.1, the authors parameter is randomly generated, which is the default approach of WS-TAXI for string type when no value is available. However, randomly generated string, such as KOVjot... below, are not very useful for testing purposes. To overcome this problem, WS-TAXI can derive more meaningful values from a populated database, when available.

Listing 5.1: Generated AddToCart SOAP envelope message

```
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:lib="http://www.example.org/Lib/">
  <soapenv:Header/>
  <soapenv:Body xmlns="http://www.example.org/Lib/">
    <SearchRequest>
      <authors>KOVjotMZBEfbeynkhtAviBIEs</authors>
    </SearchRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

In our approach, it is both necessary and reasonable to assume that, for some of the WSDL input parameters, a set of meaningful values, called an *instance pool* [30], is available. For example, in the case of Connect, it is necessary to use a correct pair of user and password. Typically, the credentials to access a WS are provided when making the registration for using it, as done for the *Amazon e-commerce service*. Thus, we assume to have an instance pool of valid user/password pairs. Other instance pools of different nature can be reasonably provided by an application user or a domain expert. For instance, it is easy to produce a list of books probably contained in Amazon. Wrapping up, if an instance pool is available for some operations, StrawBerry exploits this useful piece of information feeding the WS-TAXI database. For the WS_Lib example, we provide the instance pool for Connect and Search, as shown in Table 5.1. Back to Listing 5.1, the authors parameter can be now taken directly from the instance pool in Table 5.1, thus producing more realistic test cases.

Operation	Input Data	Operation	Input Data
Connect	u: Antonella; p: anto	Search	auth: Jean-Paul Sartre
	u: Massimo; p: Max		auth: R. Sennett, J. Cobb
	u: Paola; p: paolina		auth: Noam Chomsky
	u: Patrizio; p: P@		auth: J. David Salinger
	...		...

Table 5.1: Instance pools

Activity 4: Dependencies automaton refinement through testing.

The goal of this activity is to validate and possibly refine the dependencies automaton against the WS implementation. The test cases are selected so to cover all the dependencies; the oracle is provided by the WS implementation, as explained below. Note that since we start from the saturated automaton and the objective of the testing is to prune the false dependencies, coverage driven test selection in this case fulfills completely the purpose, i.e., we are sure we cannot miss any dependency (contrary to the well-known risk of missing functionalities in code coverage testing).

[4] eviware soapUI: http://www.soapui.org

When we invoke the WS, we cannot know in advance what the expected answer will be. However, we can always assume that for each test invocation, the WS can either return some output values or answer the request by providing an error message. We refer to the two cases as a *regular answer* and an *error answer*, respectively. The problem we have to face now is that, without analyzing the semantics of the message response it is not possible to distinguish between responses to malformed requests (e.g., a wrong cart) and negative responses to well-formed requests (e.g., a search of a book not contained into the DB). Obviously, it is always possible to define an oracle specific for the considered WS that contains the semantics of errors as can be inferred from the WS documentation. The advantage of this solution is a precise oracle while the disadvantage is that it must be built for each WS. For this reason, in this chapter we propose a partial, but general, oracle that is based on the following observations: (i) whenever invoking different operations with wrong input data, the error answer message is (almost) always the same; (ii) error answers are typically short; (iii) regular answers are typically different from each other; (iv) regular answers are typically long. This partial oracle can be realized by using a statistical approach to partition WS responses into regular and error answers. In this chapter, we do not discuss the actual implementation of such a general oracle.

The testing activity is organized into three steps. StrawBerry runs positive tests in the first step and negative tests in the second step. Positive test cases reproduce the elicited data dependencies and are used to reject fake dependencies: if a positive test invocation returns an error answer, StrawBerry concludes that the tested dependency does not exist. Negative test cases are instead used to confirm uncertain dependencies: StrawBerry provides in input to the sink operation a random test case of the expected type. If this test invocation returns an error answer, then StrawBerry concludes that the WS was indeed expecting as input the output produced by the source operation, and it confirms the hypothesized dependency as certain. If uncertain dependencies remain after the two steps, StrawBerry resolves the uncertainty by assuming that the hypothesized dependencies do not exist. Intuitively, this is the safest choice, given that at the previous step the invoked operation accepted a random input. Alternatively, we could investigate further by launching more test cases and making a statistic inference from the observed results. An empirical evaluation of the impact of this third step, and a possible improvement of StrawBerry with a significance test for the uncertain dependencies that reach the third step, is left to future work.

Step 4.1, false dependencies elimination: each *uncertain* dependency in every node is tested. For example, considering the dependency Connect.userID $\mapsto_{string}$ Search.isbn in Connect, StrawBerry executes a test for it by invoking Search passing as isbn the value of userID obtained as result of Connect on an instance pool data. It gets an error answer and therefore it removes this dependency. After this step, all dependencies whose test case produced an error message are eliminated. When deleting the last dependency that participates in a connection between two nodes, also the arc between these two nodes must be removed. Nodes that have no incoming and outgoing arc can be removed. For the nodes, different from Env, that have outgoing arcs and no incoming arc except for loops, StrawBerry adds an incoming arc from Env and adds the corresponding *certain* dependencies into Env. Note that Env can still contain *uncertain* dependencies.

Focusing on our example, all the dependencies in Env that have MakeAnOrder and Disconnect as sink operations are removed (as the corresponding arcs). Thus, the survived dependencies into Env are:

$\bigstar\mapsto_{string}$AddToCart.itemId,
$\bigstar\mapsto_{string}$Search.p, p$\in${authors,isbn,keywords,title},
$\bigstar\mapsto_{string}$Connect.p, p$\in${user,password}.

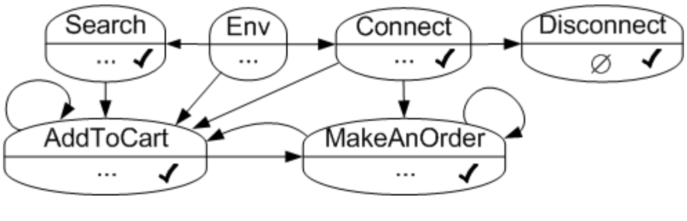


Figure 5.4: Dependencies automaton after Step 4.1

All the uncertain dependencies except for the ones stored in Env are removed. Thus, some arcs shown in Figure 5.3 are removed leading to the automaton shown in Figure 5.4. A node is validated when it stores either only *certain* dependencies or no dependency. After this step, the only non-validated node is Env as shown in Figure 5.4 where validated nodes are marked with ✓. Validation in this step is essentially due to the good functioning of the heuristics.

Step 4.2, true dependencies confirmation: this step performs a first trivial check. Env is marked as validated and all its dependencies become *certain*. By considering the automaton shown in Figure 5.4, this means that we can conclude the testing activity. However, if we had applied StrawBerry without heuristics, we would have had for instance that Connect.userID $\mapsto_{string}$ Disconnect.userID (which has been promoted, in Step 4.1, as *certain* by Heuristic 2) could not be deleted since the test did not fail, and therefore Connect would have not been validated. In this case, StrawBerry exercises every remaining uncertain dependency in every node through a negative test. For example, it executes a test for the dependency Connect.userID $\mapsto_{string}$ Disconnect.userID. By providing as input to the Disconnect operation a randomly generated input of type String, StrawBerry gets an error answer and therefore it promotes to *certain* this dependency. After this step, all dependencies whose negative test case produced an error answer are confirmed as certain.

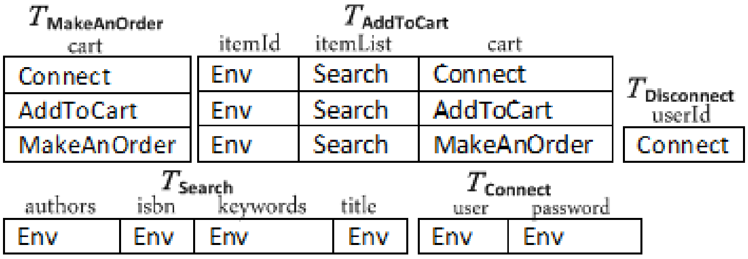
Step 4.3, solving remaining uncertain dependencies: dependencies, if any, that remain uncertain after steps 4.1 and 4.2 refer to cases in which the testing of the sink operation of a dependency did not distinguish between the output produced by the source operation or a random input. In such (experimentally few) remaining cases, StrawBerry resolves the uncertainty by assuming that the hypothesized dependency does not exist.

Activity 5: Behavior protocol synthesis.

This activity takes as input the validated dependencies automaton. For each operation op in the automaton, this activity takes into account the operations that are required to produce the input parameters of op. For instance, for AddToCart, the validated dependencies where AddToCart is a sink operation are:

$\bigstar\mapsto_{string}$AddToCart.itemId,
Connect.cart$\mapsto_{BookCart}$AddToCart.cart,
AddToCart.cart$\mapsto_{BookCart}$AddToCart.cart, and
Search.bookDetailsList$\mapsto_{BookDetailsList}$AddToCart.itemList.

By looking at these dependencies, this activity elicits that, in order to invoke AddToCart, itemId must be provided by the client, cart can be set by the output cart of either Connect, AddToCart itself, or MakeAnOrder, and itemList is set by the output bookDetailList of Search. In Figure 5.5, we graphically represent the operations that should be invoked before invoking AddToCart (see table $T_{AddToCart}$) according to the dependencies validated on its input parameters itemId, itemList, and cart. An analogous process is performed for the other operations hence leading to produce the information graphically represented in Figure 5.5.



$T_{\text{MakeAnOrder}}$

cart
Connect
AddToCart
MakeAnOrder

$T_{\text{AddToCart}}$

itemId	itemList	cart
Env	Search	Connect
Env	Search	AddToCart
Env	Search	MakeAnOrder

$T_{\text{Disconnect}}$

userId
Connect

T_{Search}

authors	isbn	keywords	title
Env	Env	Env	Env

T_{Connect}

user	password
Env	Env

Figure 5.5: Operation invocation dependencies

This information is used to synthesize an automaton that models the behavior protocol of the WS, i.e., the interaction protocol that a client has to abide by to correctly interact with the WS. In Figure 5.6, we show this automaton for our explanatory example. This automaton explicitly models also the data that has to be passed to the WS operations. Each arc label follows the syntax: operation_name '(' comma_separated_inputs ')' ':' comma_separated_outputs. The synthesis algorithm reflects the validated data dependencies in conjunction with the operation invocation dependencies represented in Figure 5.5. The algorithm is presented in Section 5.3. For the sake of readability, in Figure 5.6, we omit I/O data for some operation and in place of a data parameter name we use its initials.

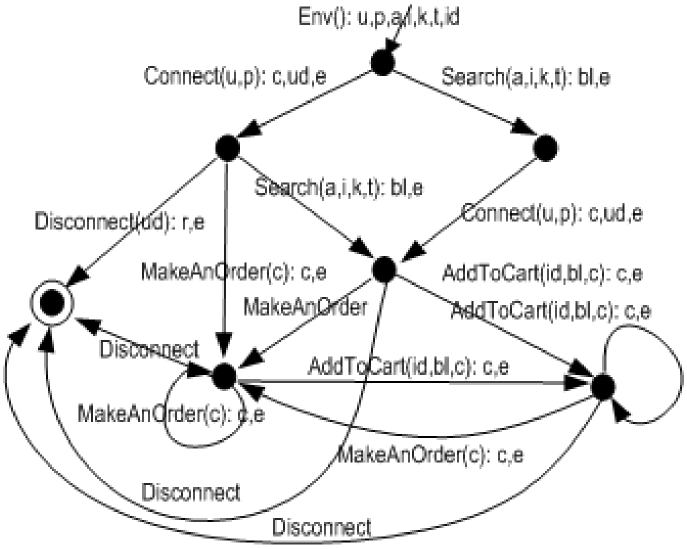


Figure 5.6: Behavior protocol automaton

In Figure 5.6, the state with the (no-source) incoming arrow and the doubled circled state are the initial and final states, respectively. Note that, in general, the WSDL of the WS can define operations that are not taken into account by the validated dependencies automaton since these operations are not involved in any dependency because they can be always invoked. In order not to complicate a behavior protocol automaton, this aspect is reflected by implicitly considering that these operations become loop transitions on every state of the automaton.

5.3 Method Formalization

In this section we formalize the StrawBerry method. This formalization rigorously defines all the method stages concluding with a detailed presentation of the StrawBerry testing process and of the behavior protocol automaton synthesis. Furthermore, it represents the specification from which the prototypal implementation of StrawBerry has been realized. For the sake of simplicity, we omit the formalization of the three heuristics discussed in Section 5.2.3 since it is straightforward.

Let W be a WSDL interface, we denote with $\mathrm{Op_W}$ the set of all the operation names of W.

We denote with $\mathfrak{D}_W$ the set of all I/O data dependencies of W obtained by syntactically matching the type of an output parameter of an operation in $\mathrm{Op_W}$ with the type of an input parameter of another operation in $\mathrm{Op_W}$. $\mathfrak{D}_W$ can be partitioned into C_W and U_W that denote the set of all the *certain* and *uncertain* dependencies, respectively. Thus, with either $op.p \mapsto\!\!\!\!\checkmark_t\ op'.p'$ or $\bigstar \mapsto\!\!\!\!\checkmark_{t'} op'.p'$ we denote elements of C_W for some $op{,}op' \in Op_W$ and parameter names $p{,}p'$ of type t. Analogously, with either $op.p \mapsto_t op'.p'$ or $\bigstar \mapsto_{t'} op'.p'$ we denote elements of U_W. Note that this notation implies that p is the name of an output parameter of op and p' is the name of an input parameter of op'.

Hereafter if $op.p \mapsto_t op'.p'$ ($op.p \mapsto\!\!\!\!\checkmark_t\ op'.p'$), we write that "**a dependency exists**" for op. We can also write that op' "**depends on**" op. If $\bigstar \mapsto_{t'} op'.p'$ ($\bigstar \mapsto\!\!\!\!\checkmark_{t'} op'.p'$), we write that op' "**is dependent**" on the environment.

Once the set of I/O dependencies has been built we can construct the *Dependencies Automaton* as defined in Def. 17. This definition makes use of the function *Node generator* defined in Def. 16. The role of this function is to define the nodes of the automaton that will be built by Def. 17. A special case is the node Env that is directly added by a saturation rule, see Def. 18. As described in Section 5.2 *Node generator* implements the *Step 2.1, node generation* of the *Activity 2*. The operations for which a node must be built are identified by means of the *I/O Dependency set* $\mathfrak{D}_W$.

Definition 16 (Node generator)
Node generator N_{gen}:$\mathrm{Op_W}{\longrightarrow}2^{\mathfrak{D}_W}$*, is a function that given as input* $op{\in}Op_W$ *returns the set* $\mathrm{D}{\in}2^{\mathfrak{D}_W}$ *s.t. either* D *is empty or for each* $\mathrm{dep}{\in}\mathrm{D}$*, a* dependency exists *for* op *and there does not exist* $\mathrm{D}'{\in}2^{\mathfrak{D}_W}$ *s.t. for each* $\mathrm{dep}'{\in}\mathrm{D}'\backslash\mathrm{D}$*, a* dependency exists *for* op*.*

At the beginning, all the dependencies stored into a node generated by *Node generator* are *uncertain*. `StrawBerry` makes use of the previously discussed three heuristics in order to set to *certain* some dependencies and to remove some others. We recall that by exploiting the *Node generator* function and the *I/O Dependency set*, `StrawBerry` synthesizes an automaton that models all the chains of data dependencies that should be taken into account while using the WS. Each arc from a node to another node reflects the I/O Dependency set, thus we call this automaton the *Dependencies automaton*.

Definition 17 (Dependencies automaton)
A Dependencies automaton $A_W{=}(N,\Delta)$ *of a WSDL interface* W *is an automaton where:*
▶ $N{=}\{(op_1,N_{gen}(op_1)),\cdots,(op_v,N_{gen}(op_v))\}$ *is the set of nodes s.t.* $\{op_1,\cdots,op_v\} \subseteq \mathrm{Op_W}$ *and for each* $i = 1, \cdots, v$ *either a* dependency exists *for* op_i *or* op_i depends on *some operation;*
▶ $\Delta \subseteq N \times N$ *is the set of arcs s.t.* $\Delta{=}\{(n_{op'_1},n_{op_1}), \cdots,(n_{op'_j},n_{op_j})\}$ *and for each* $i = 1,\cdots,j$*, then* $(n_{op'_i},op_i,n_{op_i}){\in}\Delta$ *iff a* dependency exists *for* op_i, op_i depends on op'_i, $n_{op_i}{=}(op_i,N_{gen}(op_i))$*, and* $n_{op'_i} = (op'_i,N_{gen}(op'_i))$*.*

As already mentioned in Section 5.2, we need to “saturate” the automaton in order to complete it with respect to the possibility for the environment to directly provide input parameters. The result of this saturation step is called the *Saturated dependencies automaton*.

Definition 18 (Saturated dependencies automaton)
Let $A_W = (N,\Delta)$ *be the* Dependencies automaton *of a WSDL interface* W*, the* Saturated dependencies automaton S_W *of* W *is the tuple* (N_{sat},Δ_{sat}) *where:*
▶ $N_{sat} = N \cup \{n_{Env}\}$ *s.t.* $n_{Env}{=}(Env,\mathrm{D})$ *is the environment node and* $\mathrm{D}{=}\{\bigstar{\mapsto_t}op.p \mid op \in \mathrm{Op_W},\ p$ *of type* $t\}$*;*
▶ $\Delta_{sat} = \Delta \cup \Delta_{Env}$, $\Delta \cap \Delta_{Env} = \emptyset$*, s.t.* $\Delta_{Env} = \{(n_{Env},n_{op_1}),\cdots,(n_{Env},n_{op_j})\}$ *and for each* $i = 1,\cdots,j$*, then* $(n_{Env},n_{op_i}) \in \Delta_{Env}$ *iff a* dependency exists *for* op_i*, and* $n_{op_i} = (op_i,N_{gen}(op_i))$*.*

Definition 19 (I/O dependencies chain)
Let $S_W{=}(N_{sat},\Delta_{sat})$ *be the saturated dependencies automaton of a WSDL interface* W*, an* I/O dependencies chain *of* S_W *is a* $c{\in}N^*_{sat}$ *defined in such a way that there exists* $m{>}0$, $n_0,\cdots,n_m{\in}N_{sat}$ *s.t.* $n_0{=}(Env,\mathrm{D})$, $c{=}\langle n_0 n_1 \cdots n_m\rangle$*, and* $(n_0, n_1){\in}\Delta_{sat},\cdots,(n_{m-1}, n_m){\in}\Delta_{sat}$*.*

Let $S_W{=}(N_{sat},\Delta_{sat})$ be a saturated dependencies automaton, given a node $n{\in}N_{sat}$, the set of I/O dependencies chains leading to n (and originating from the node of name Env) is denoted as $Ch(n)$. We denote the *normalization* of $Ch(n)$ with $\overline{Ch(n)}$ and it is defined as the set of traces of $Ch(n)$ without either loops (i.e., loop transitions) or cycles (i.e., cyclic paths).

Note that $\overline{Ch(n)}$ is a finite set, whereas $Ch(n)$ can be infinite.

Given $(op,D_{op}){\in}N_{sat}$ and $op{\neq}Env$, with $IP(op)$ we denote the set of instance pools for the operation of name op. That is $IP(op){=}\{(p_1{:}v_1,\cdots,p_n{:}v_n)$ *s.t.* $p_1,\cdots,p_n$ *are input parameters of* op *and* $v_1,\cdots,v_n$ *are the values of* $p_1,\cdots,p_n$*, respectively*$\}$. With $SoapEnv$ we denote the set of all the SOAP messages that conform to the XML Schema of W. We denote the oracle that we use for testing purposes as a function $\mathcal{O}racle : SoapEnv{\longrightarrow}\{regular,error\}$. In the following, we use a function $\mathcal{T}est_W : SoapEnv{\longrightarrow}SoapEnv$ that represents the execution of a test case (encoded as a SOAP message) on a WS implementing W. That is, it represents a WS operation invocation (i.e., the operation input message) retrieving another SOAP message as answer (i.e., the operation output message). We also use a function $\mathcal{R}esp2\mathcal{R}eqs : SoapEnv{\times}\mathrm{Op}_W{\times}\mathrm{Op}_W{\longrightarrow}SoapEnv^*$ that takes as input the response of the invocation of $op{\in}\mathrm{Op}_W$ and returns a tuple of requests for $op'{\in}\mathrm{Op}_W$ that depends on op. Thus, each of these requests is built by taking into account the dependencies stored in the node of op. Listing 5.2 is an operational description of the testing procedure that `StrawBerry` performs to produce the validated dependency automaton out of the saturated one. Note that this description is not the optimal algorithm with respect to computational load. However optimality is not the focus here. This procedure exploits the $\mathcal{O}racle$, $\mathcal{T}est_W$, and $\mathcal{R}esp2\mathcal{R}eqs$

functions. The validated dependency automaton, as synthesized by our testing procedure, is defined by Def. 20. In Listing 5.2, given an operation $op \in \mathrm{Op}_\mathrm{W}$, we denote the node of op in N_{sat} as $node(op)$. Furthermore, we denote a SOAP envelope message as either $soap$ or $soap_i$ for some i.

Listing 5.2: `StrawBerry` **testing procedure**

```
$(N,\Delta)$ being the $Saturated\ dependencies\ automaton$ of a WSDL interface $W$, perform
   the following steps:
Inizialization: mark every dependency in $S_W$ as nonVisited;
                create an empty stack called $Stack$;
Step 1: while $\exists\ v=(op, D_{op}) \in N$ that stores a nonVisited uncertain dependency do
   while $\exists\ ch=\langle op_1 \cdots op_n op_{n+1}\rangle \in \overline{Ch(op)}$ $(op=op_{n+1})$ s.t. $node(op_1) \cdots node(op_{n-1})$ store only
         certain dependencies and $\exists$ in $node(op_n)$ a nonVisited dependency, $op_{n+1}$
       depends on, do
     if $IP(op_1) \neq \emptyset$ then produce $soap_1$ from $IP(op_1)$;
       else produce $soap_1$ randomly for $op_1$;
     push $\mathcal{R}esp2\mathcal{R}eqs(\mathcal{T}est_W(soap_1), op_1, op_2)$ into $Stack$;
     set $i$ to $2$;
     while $i < n+1$ do
       foreach $soap$ popped out from $Stack$ do
         push $\mathcal{R}esp2\mathcal{R}eqs(\mathcal{T}est_W(soap), op_i, op_{i+1})$ into $Stack$;
       set $i$ to $i+1$;
   foreach $soap$ popped out from $Stack$ do
     if $\mathcal{O}racle(\mathcal{T}est_W(soap))=error$ then remove from $node(op_n)$ all dependencies $op_{n+1}$
         depends on w.r.t. all the output parameters $p$ of $op_n$ that are involved
          in $soap$
       else mark as visited these dependencies;
   if $node(op_n)$ stores no dependency $op_{n+1}$ depends on then remove $(op_n, op)$ from $\Delta$
Step 2: if $\exists\ v=(op, D_{op}) \in N$, $op \neq Env$ and $v$ has no incoming arc then add $(s_{Env}, v)$ to
      $\Delta$, $s_{Env}=(Env, D_{Env})$, and add the corresponding certain dependencies to
    $D_{Env}$;
   foreach $v=(op, D_{op}) \in N$ that stores an uncertain dependency do
     foreach $op'$ that depends on $op$ w.r.t. an uncertain dependency do
       produce $soap$ randomly for $op'$;
       if $\mathcal{O}racle(\mathcal{T}est_W(soap))=error$ then mark as certain all dependencies in
           $node(op)$, $op'$ depends on;
Step 3: foreach $v=(op, D_{op}) \in N$ s.t. $op \neq Env$ do
   remove all the uncertain dependencies from $D_{op}$;
   if $D_{op}=\emptyset$ and $v$ has no incoming arc then remove $v$.
```

Definition 20 (Validated dependency automaton)
The Validated dependency automaton V_W *of a WSDL interface* W *is the pair* (N,Δ) *that holds the following properties:*

- $\forall n \in N$: $\exists op \in \mathrm{Op}_\mathrm{W}$: $n=(op, D_{op})$ *and either* D_{op} *contains only* certain *dependencies or it is empty*;
- $\forall n \in N$: $Ch(n) \neq \emptyset$;
- $\forall d \in \Delta$*:* $\exists op, op' \in \mathrm{Op}_\mathrm{W}$: $d=((op, D_{op}), (op', D_{op'})) \wedge op.p \not\mapsto_t op'.p' \in D_{op}$.

From the *Validated dependency automaton*, the transformations specified in Def. 22 produce a *Behavior protocol automaton*. Let $V_W=(N,\Delta)$ be the *Validated dependency automaton* of a WSDL interface W, with $\mathrm{ioDS}(V_W)$ we denote the I/O dependency set of V_W and with $\mathrm{Op}(V_W)$ the set of operation names for V_W. $\mathrm{ioDS}(V_W)$ corresponds to the set of I/O dependencies stored in the nodes of V_W. Note that they are all certain dependencies. $\mathrm{Op}(V_W)$ corresponds to the set of operation labels stored in the nodes of V_W, including `Env`. Starting from V_W `StrawBerry` produces a table T_{op} for each $op \in \mathrm{Op}(V_W)$ different from Env. $T_{op}=\{(o_1, \cdots, o_n) \in \mathrm{Op}(V_W)^n$ *s.t.* n *is the number of input parameters of* op *and for each parameter* p *of* op, *an operation* o_i *exists s.t.* $o_i.p' \not\mapsto_t op.p \in ioDS(V_W)\}$.

By taking into account each T_{op}, StrawBerry produces a set Υ of sets of operations for which no mutual dependency exists. Each set of operations in Υ corresponds to a connected component in the behavior protocol automaton to be synthesized.

For example, $\{op,op'\}\in\Upsilon$ means that neither $op.p \not\mapsto^{*}_{t} op'.p'$ nor $op'.p' \not\mapsto^{*}_{t} op.p$ hold for any p, p', and we say that op and op' are independent. Thus four states, s_1,s_2,s_3, and s_4, and four transitions, (s_1,op,s_2), (s_2,op',s_4), (s_1,op',s_3), and (s_3,op,s_4), are produced in the behavior protocol automaton. In general, if $op_1, \cdots, op_n$ $(n > 1)$ are independent, then $n + (\sum_{i=1}^{n-1} \frac{n!}{i!}) + 2$ states are generated and sequences of transitions labelled with $op_1, \cdots, op_n$ are produced among these states in order to build all the linearizations modeling the interleaving of $op_1, \cdots, op_n$. If $\{op\}\in\Upsilon$, the produced connected component is represented by the transition $(s1, op, s2)$ and the states $s1$ and $s2$. Note that each of these connected components has a source state and a sink state. For the sake of presentation, in Def. 22 we use a function, CCB (Connected Component Builder), that takes as input V_W and produces the set $\{k_1,\cdots,k_h\}$ of above discussed connected components. We denote with k_i^{source} and with k_i^{sink} the source state and the sink state of k_i, respectively. Def. 22 uses the definition of trace for a behavior protocol automaton (see Def. 21).

Definition 21 (Trace)
Let I_W=(S,F,s_0,A,Δ) be a behavior protocol automaton, a trace *of I_W is a $t\in A^*$ defined in such a way that there exist $n>0$, $s_0,\cdots,s_n\in S$ such that $t=\langle o_1 o_2 \cdots o_n\rangle$ and $(s_0, o_1, s_1)\in\Delta,\cdots,(s_{n-1}, o_n, s_n)\in\Delta$.*

Let I_W=(S,F,s_0,A,Δ) be a behavior protocol automaton, given a state $s\in S$, the set of traces leading to s (and originating from s_0) is denoted as $Tr(s)$.

Definition 22 (Behavior protocol automaton)
Let $V_W = (N,\Delta)$ be the Validated dependencies automaton *of a WSDL interface W, the* Behavior protocol automaton *of W is the tuple (S,F,s_0,A,Δ') where:*
▶ *S= $\{s|s$ is a state of a connected component $k\in CCB(V_W)\}$.*
▶ *F= $\{s_F^1,\cdots,s_F^m\}$ where each s_F^i is the sink state of a connected component built from operations that are not source operations of any dependency.*
▶ *s_0= k^{source} where $k\in CCB(V_W)$ is built from only Env.*
▶ *A=$\{op(p_1,\cdots,p_m)$:$o_1,\cdots,o_k$ s.t. $op\in$ $\mathrm{Op_W}$, $o_1,\cdots,o_k$ are output parameters of op and $p_1,\cdots,p_m$ are input parameters of $op\}$.*
▶ *$\Delta'\subseteq S\times A\times S$, Δ'=$\Delta_{cc}\cup\Delta''\cup\Delta_{loop}$, and $\Delta_{cc}\cap\Delta''\cap\Delta_{loop}=\emptyset$, where:*
▼ *Δ_{cc} is the union set of the sets of transitions of each connected component $k\in CCB(V_W)$.*
▼ *Δ''=$\{(s,l,s')$ | there exist $op\in\mathrm{Op}(V_W)$ and $k\in CCB(V_W)$ such that: s'=k^{sink} and k contains a transition labelled with op; and for each $tr=\langle op_1\cdots op_n\rangle\in Tr(s)$ and each $(o_1,\cdots,o_m)\in T_{op}$ s.t. o_i contained in tr, then l=$op(p_1,\cdots,p_m)$:$out_1,\cdots,out_k$ where $out_1,\cdots,out_k$ are output parameters of op, for each i=$1,\cdots,m$ then p_i is an output parameter of o_i, and $o_i.p_i \not\mapsto_t op.p \in \mathrm{ioDS}(V_W)$ for some p that is input parameter of $op\}$.*
▼ *Δ_{loop}=$\{(s,l,s)$ | for all $s\in S$, $op\in\mathrm{Op_W}\backslash\mathrm{Op}(V_W)$, l=$op(p_1,\cdots,p_m)$:$o_1,\cdots,o_k$ where $o_1,\cdots,o_k$ are output parameters of op and $p_1,\cdots,p_m$ are input parameters of $op\}$.*

5.4 The Amazon E-Commerce Service Case Study

In this section, we show the results of the application of StrawBerry to an existing WS, that is the *Amazon E-Commerce Service* (AECS). AECS is part of the Amazon Associates WS suite[5]. The aim of this section is to show the applicability of StrawBerry to a complex WS that is well-known and widely used by practitioners. Moreover, AECS is well-documented. Thus it allowed us to validate that the concepts underlying StrawBerry and the choices we made to realize them are reasonable in practice.

We focus on discussing the collected results rather than on detailing the execution steps of StrawBerry to AECS. This would be impractical due to the size of AECS in terms of I/O data dependencies. Starting from the AECS WSDL[6], StrawBerry performs the steps described in Section 5.2.3 producing the results discussed below.

[5] Amazon Associates: http://aws.amazon.com/associates/

[6] AECS WSDL: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

Operation	*certain* after 1.2	*uncertain* after 1.2	*uncertain* after 4.1	*uncertain* after 4.2
Help	2	2358	0	0
ItemSearch	44	2838	18	0
ItemLookup	44	2838	14	0
BrowseNodeLookup	2	1572	0	0
ListSearch	46	11222	8	0
ListLookup	46	11222	4	0
CustomerContentSearch	46	15676	4	0
CustomerContentLookup	46	15676	8	0
SimilarityLookup	44	2838	12	0
SellerLookup	2	5502	10	0
CartGet	2	4978	32	0
CartAdd	2	4978	24	0
CartCreate	2	4978	16	0
CartModify	2	4978	32	0
CartClear	2	4978	16	0
TransactionLookup	2	5502	6	0
SellerListingSearch	2	8908	10	0
SellerListingLookup	2	8908	12	0
TagLookup	46	11222	38	0
VehicleSearch	2	1310	20	0
VehiclePartSearch	46	3886	4	0
VehiclePartLookup	46	3886	4	0
MultiOperation	90	35020	75	0
Total:	568	175274	367	0

Table 5.2: Summary of the AECS case study results

Step 1.1: StrawBerry discovers that AECS exports 23 operations and elicits 187894 dependencies. Table 5.2 summarizes the discovered data; the operations are listed as they appear in the WSDL of AECS.
Step 1.2: heuristics 1 and 2 allow StrawBerry to promote 568 dependencies as *certain* (40 and 528, respectively). The AECS reference guide allows us to enable heuristic 3 since it reports that all the operation parameters of type Errors are used to encode error answers. Thus, Heuristic 3 removes 12052 dependencies. In Table 5.2 we show how the 568 discovered *certain* dependencies are distributed among the AECS operations. At the end of this step, there remain 175274 *uncertain* dependencies.
Step 2.1: StrawBerry generates 23 nodes, one for each operation, and each of them stores a number of *certain* and *uncertain* dependencies as reported by the first three columns of Tab. 5.2.
Step 2.2: StrawBerry generates a dependency automaton that has 23 nodes and 529 arcs. The size of this automaton prevents us to graphically show it. However, in Figure 5.7, we show an excerpt from the behavior protocol automaton constructed by StrawBerry at the end of the process.
Step 2.3: the previous automaton is saturated by adding the Env node and 23 arcs, each of them from Env to another node. Env stores 350 *uncertain* dependencies.

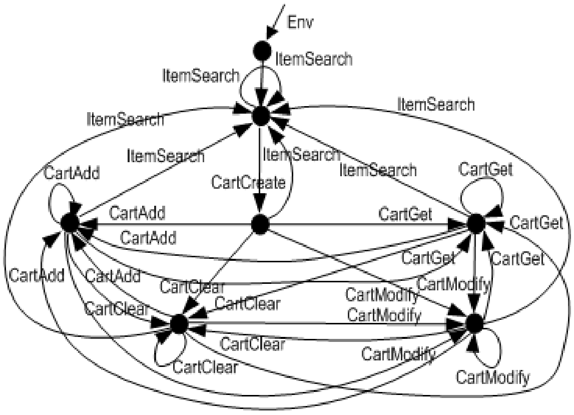


Figure 5.7: An excerpt from the behavior protocol of AECS

Activity 3: we build an instance pool in order to generate, by means of StrawBerry, the SOAP envelope messages for testing AECS. We need to provide SOAP test messages with a pair of unique identifiers that are required, for security purposes, by each operation. These identifiers are provided by Amazon after the *Amazon Associate* registration process. Furthermore, we can add instance pools related to meaningful

Amazon items, e.g., some author names as shown in Tab. 5.1.

Step 4.1: the previously generated test cases are used by StrawBerry to prune the set of 175274 *uncertain* dependencies obtained after Step 1.2. After Step 4.1, in which we test each of them, only 367 *uncertain* dependencies survive, distributed among the operations as shown in the fourth column of Tab. 5.2. Note that, already after Step 4.1, we can have some operations (Help and BrowseNodeLookup) for which no *uncertain* dependency survives.

Step 4.2: the surviving 367 dependencies are confirmed as *certain* by the tests in Step 4.2 and hence there is no *uncertain* dependency to be solved in Step 4.3. Thus, StrawBerry directly performs Activity 5.

Activity 5: starting from the validated dependency automaton, as obtained after the execution of Step 4.2, StrawBerry synthesizes the behavior protocol automaton of AECS. The validated dependency automaton has 24 nodes and 288 arcs. In Figure 5.7, we show an excerpt concerning all the "item search" and "cart management" operations of AECS. For the sake of readability we omit the data parameters in the operation labels. Furthermore, for each state there are other incoming and outgoing transitions from and to states that do not appear in the figure. By looking at Figure 5.7 one could think that one specific state is corresponding to one specific operation; however as already seen in Figure 5.6, this does not hold in general.

We performed some ad hoc validation of the synthesized automaton. We checked that all the results described above and the synthesized protocol match with what is described in the AECS API reference[7]. Moreover we further validated the synthesized behavior protocol of AECS through a Web client provided by Amazon[8]. Thus, we empirically verified that StrawBerry produces a realistic model for AECS.

Besides showing the effectiveness of StrawBerry, this case study highlights that, even when the necessary information is available, the hand-made provisioning of the behavior protocol is a difficult and error prone task.

5.5 Related Work

Several authors have recently addressed the problem of deriving a behavioral model from an implemented system. We discuss here some of these works, including work in WP4 in the domain of state machine learning algorithms, with respect to the StrawBerry method.

In [44], the authors describe a technique, called GK-Tail, to automatically generate behavioral models from (object-oriented) system execution traces. GK-Tail assumes that execution traces are obtained by monitoring the system through message logging frameworks. For each system method, an Extended Finite State Machine (EFSM) is generated. It models the interaction between the components forming the system in terms of sequences of method invocations and data constraints on these invocations. The correctness of these data constraints depends on the completeness of the set of monitored traces with respect to all the possible system executions that might be infinite. Furthermore, since the set of monitored traces represents only positive samples of the system execution, their approach cannot guarantee the complete correctness of the inferred data constraints. Instead the set of data dependencies, inferred by StrawBerry, concerns both positive and negative samples and it is syntactically correct by construction. However, it might not be correct semantically since it may contain false positives. These false positives are detected by the testing phase. Furthermore, dealing with black-box WSs, we cannot assume to take as input a set of interaction traces. Finally note that StrawBerry is an extra-procedural method, whereas GK-Tail is intra-procedural. In fact we synthesize a model of the possible interactions between the WS and its environment, whereas they synthesize an intra-system interaction model.

The work described in [30] (i.e., the SPY approach) aims to infer a formal specification of stateful black-box components that behave as data abstractions (Java classes that behave as data containers) by observing their run-time behavior. SPY proceeds in two main stages: first, SPY infers a partial model of the considered Java class; this partial model is generalized to deal with data values beyond the ones specified by the given instance pools. The model generalization is based on two assumptions: (i) the value of method parameters does not impact the implementation logic of the methods of a class; (ii) the behavior observed during the partial model inference process enjoys the so called "continuity property"

[7] AECS API reference: http://awsdocs.s3.amazonaws.com/ECS/latest/aaws-dg.pdf

[8] http://www.awszone.com/scratchpads/index.aws

(i.e., a class instance has a kind of "uniform" behavior). In our context, we cannot rely on the previously mentioned assumptions.

The approach described in [83], and implemented by Jadet, analyzes Java code to infer sequences of method calls. These sequences are then used to produce object usage patterns that serve to detect object usage violations in the code. Differently from StrawBerry, Jadet is a white-box method. Furthermore, as it is for the work described in [44], Jadet focuses on modeling objects from the point of view of single methods that is a goal different from ours.

The work described in [82] (i.e., OP-Miner) is very similar to Jadet. Differently from our work, it is a white-box approach. Java code is analyzed to infer the sequence of operations an object variable goes through before being used as a parameter. In general, this is slightly similar to what StrawBerry synthesizes but, in practice, analogously to what Jadet does, this is done by looking at each single method. In this sense the analysis performed by OP-Miner (and Jadet) is intra-procedural, whereas our approach is extra-procedural.

Work in WP4 is about automated approaches for inferring state machines by observing the output that the system produces when stimulated with selected inputs [13]. The main difference between StrawBerry and the work in WP4 is that we have the opposite problem of relaxing, through testing, some data dependencies between the system operations (when its existence is not certain) rather than adding new dependencies, as it is done in the work in WP4. Furthermore, the work in WP4 allows for the inference of more detailed behavioral models that contain both intra- and extra-procedural information, whereas StrawBerry concerns the synthesis of behavior protocol with only extra-procedural information.

The authors of [49] describe a learning-based black-box testing approach in which the problem of testing functional correctness is reduced to a constraint solving problem. A general method to solve this problem is presented and it is based on function approximation. Functional correctness is modeled by pre- and post-conditions that are first-order predicate formulas. A successful black-box test is an execution of the program on a set of input values satisfying the pre-condition, which terminates by retrieving a set of output values violating the post-condition. Black-box functional testing is the search for successful tests w.r.t. the program pre- and post-conditions. As *coverage* criterion, the authors formulate a convergence criterion on function approximation. Their testing process is an iterative process. At a generic testing step, if a successful test has to be still found, the approach described in [49] exploits the input and output assignments obtained by the previous test cases in order to build an approximation of the system under testing and try to infer a valid input assignments that can lead the system to produce an output either violating the post-condition or useful to further refine the system approximated model. The testing phase of our approach shares some ideas with the approach described in [49]. That is, through black-box testing, we refine an approximated data-flow model in order to prune fake I/O dependencies. However, we do not use *function approximation theory* and our coverage criterion is established by looking at the inferred I/O dependencies.

5.6 Summary

In this chapter we have presented the StrawBerry method. It takes as input a WSDL description, matches by type the input and output parameters of its operations, applies some graph synthesis and heuristics, and going through a testing phase, eventually synthesizes what we have called the *Behavior Protocol* automaton.

StrawBerry fulfills an important exigency in CONNECT, that is to get more semantic information for a networked system (e.g., a WS), where the current practice is to publish only its signature. A behavioral model is required for the CONNECTor synthesis process to both understand how a networked system should be used, and to properly synthesize a mediator in order to achieve interoperability among heterogeneous networked systems. As already discussed at the beginning of this chapter, this exigency should be addressed by the work specific to work package WP4. However, due to the differences of the StrawBerry approach with respect to the WP4's work, we preferred to keep the work described in this chapter separate from the work of WP4.

The method that we propose is practical and realistic in that it only assumes: (i) the availability of the WSDL; and (ii) the possibility to derive a partial oracle that can distinguish between *regular* and *error* answers. This oracle is needed in the testing stage to confirm or reject uncertain dependencies. For the work described in this chapter, we have taken assumption (ii) in strict sense, in that we have assumed the existence of this oracle, i.e., that the available WS information allows a tester to recognize when a test outcome is an error message. In future work, we intend to investigate if and how assumption (ii) could be relaxed, and, where such a partial oracle does not exist, the deterministic testing steps could be replaced by a statistical testing session.

We have started to show, through its application to the the Amazon WS, that the method is viable, and that it nicely converges to a realistic automaton. We obviously need to carry out more empirical investigation to convey such preliminary evidences into a real quantitative assessment of the method. However, the case study convinced us that the combination of heuristics and basic testing can work quite effectively. In particular, the introduction of heuristics for optimization seems interesting and we believe that it is the first direction to push further to reduce the testing effort in the subsequent steps.

6 Conclusion and Future Work

Given the interaction protocols of networked systems, 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 networked systems. We recall that the role of work package WP3 is to devise automated and compositional approaches to CONNECTor synthesis, which can be performed at run-time.

In this deliverable we have presented a formal theory sustaining the automated synthesis of application- and middleware-layer protocol mediators. We have formalized the concept of *mediating connector* (also refered to as mediator or CONNECTor) between application-layer protocols by rigorously defining two essential relationships: protocol *matching* and *mapping*. The former allows one to establish whether a mediator letting two mismatching protocols interoperate exists. The latter is essentially the algorithm that should be performed to synthesize the required mediator, when it exists.

In order to also support middleware-layer mediation, we have analyzed the different dimensions of protocol heterogeneity at the middleware-layer and we have exploited this analysis to formalize the various solutions to middleware interoperability existing in the literature. This formalization further allows us to characterize to which extent the synthesis method defined for application-layer mediators can be applied to middleware-layer mediation, hence hinting on the adjustments to be applied to the devised theory.

Since our work on automated mediation is based on the assumption that a model of the interaction protocol for a networked systems is dynamically discovered, we have finally presented an approach, based on data-flow analysis and testing, to the automated elicitation of application-layer protocols from software implementations. This work allowed us to reason about the requirements that should be satisfied, in favor of the synthesis method, by the work conducted within work package WP4 [3].

As future work, we intend to rigorously evaluate the theory underlying application-layer mediator synthesis with respect to the classification of possible mismatches informally discussed in Section 3.5. To this end, we plan to formalize a correctness proof of the theory with respect to the devised mismatches. A limit of the current approach is that we do not address data mismatches. We also plan to investigate and extend our approach in this direction.

The theory devised so far is able to deal with peer-to-peer protocols only. We will adjust it in order to deal with other architectural styles, such as client-server systems. The experiments conducted so far have shown that, for this purpose, we will need to define a notion of behavioral equivalence, which is dependent on the protocol role, e.g., based on *bisimulation* for peer-to-peer protocols, or on *simulation* for client-server protocols. In this direction, experimentation of the theory with systems conforming to different architectural styles will be crucial. This also necessitates extending the theory to deal with middleware-layer interoperability.

We found some differences between the model of interaction protocols as learned by the approach devised within work package WP4 and the model expected by the current mediator synthesis method. However, these differences do not seem to be severe. This suggests us that a suitable model transformation can be easily defined. As future work, we will work in conjunction with work package WP4 either to solve these differences or to define the needed transformation.

Furthermore, we intend to revise and enhance the current theory of mediating connectors in order to take into account two relevant characteristics for CONNECT: (i) Quality-of-Service dimensions of the interaction that have been not considered yet. This aspect is particularly crucial for middleware-layer mediation. (ii) Compositionality of the mediator synthesis method. This aspect increases the efficiency of the synthesis process hence allowing its execution at run-time.

Concerning the protocol elicitation approach described in Chapter 5, as future work, we intend to investigate if and how the *oracle assumption* could be relaxed.

Finally, for all the approaches described in this deliverable, we obviously need to carry out more empirical investigation to convert the achieved preliminary results into a real quantitative assessment of the various methods.

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 D4.1: Establishing basis for learning algorithms. FET IP CONNECT EU project, FP7 grant agreement number 231167, http://connect-forever.eu/.

[4] Jabber Software Foundation, http://www.jabber.org/.

[5] Windows Live Messenger, http://www.messenger.it/.

[6] WSDL: Web Services Description Languages v1.1 spec. http://www.w3.org/tr/2001/note-wsdl-20010315.

[7] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[8] M. V. S. Andrew S. Tanenbaum. *Distributed systems : principles and paradigms*. Upper Saddle RIiver, NJ : Pearson Prentice Hall, 2007.

[9] K. Arnold. The jini architecture: Dynamic services in a flexible network. In *DAC*, pages 157–162, 1999.

[10] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: a WSDL-based testing tool for Web Services. In *ICST 2009, Denver, Colorado - USA*. IEEE, 2009.

[11] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for Playing Games! In *CAV 2007*, 2007.

[12] 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.

[13] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *FASE 2008, Budapest, Hungary*, pages 317–331, 2008.

[14] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In *ESEC/FSE09*, 2009.

[15] P. Bidinger, A. Schmitt, and J.-B. Stefani. An abstract machine for the kell calculus. In *FMOODS*, pages 31–46, 2005.

[16] P. Bidinger and J.-B. Stefani. The kell calculus: Operational semantics and type system. In *FMOODS*, pages 109–123, 2003.

[17] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *ICSOC 2006, Chicago, USA*, 2006.

[18] Y.-D. Bromberg. *Solutions to middleware heterogeneity in open networked environment*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelynes, 2006.

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

[20] Y.-D. Bromberg and V. Issarny. Formalizing middleware interoperability: From design time to runtime solutions. Technical report, Rocquencourt, France, 2008.

[21] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi. Automatic Service Composition and Synthesis: the Roman Model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[22] C. Canala, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *FMOODS 2006*, 2006.

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

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

[25] W. Emmerich. Software engineering and middleware: a roadmap. In *ICSE - Future of SE Track*, pages 117–129, 2000.

[26] D. Fensel and C. Bussler. The web service modeling framework wsm. *Journal of Electronic Commerce Research and Application*, 1(1):113–137, 2002.

[27] C. A. Flores-Cortés, G. S. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *IEEE Distributed Systems Online*, 8(7), 2007.

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

[29] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *SFM*, volume 26, pages 1–24. Springer, 2003.

[30] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *ICSE 2009, Vancouver, Canada*, 2009.

[31] 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.

[32] E. Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80, 1999.

[33] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[34] P. Inverardi and M. Nesi. Deciding Observational Congruence of Finite-State CCS expressions by Rewriting. *Theor. Comput. Sci.*, 139(1-2):315–354, 1995.

[35] P. Inverardi and M. Tivoli. Deadlock-free software architectures for COM/DCOM Applications. *Elsevier Journal of Systems and Software*, 2003.

[36] P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. In *Springer, LNCS 2804*, 2004.

[37] M. Jeronimo and J. Weast. *UPnP Design by Example :A Software Designer's Guide to Universal Plug and Play*. Intel Press, 2003.

[38] 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.

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

[40] A. Kucera and O. Strazovský. On the Controller Synthesis for Finite-State Markov Decision Processes. In *FSTTCS 2005*, pages 541–552, 2005.

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

[42] 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.

[43] N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, T. Li, R. Boutaba, and F. Cuervo. Osda: Open service discovery architecture for efficient cross-domain service provisioning. *Computer Communications*, 30(3):546–563, 2007.

[44] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE 2008*, pages 501–510, NY, USA, 2008. ACM.

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

[46] A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[47] M.Autili, P.Inverardi, A.Navarra, and M.Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In *ICSE 2007*, 2007.

[48] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE*, pages 178–187, 2000.

[49] K. Meinke. Automated Black-box Testing of Functional Correctness using Function Approximation. *SIGSOFT Softw. Eng. Notes*, 29(4):143–153, 2004.

[50] T. Melliti, P. Poizat, and S. B. Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In *FASE 2008, LNCS 4961, Springer*.

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

[52] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge Univ. Press, 1999.

[53] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[54] S. B. Mokhtar, N. Georgantas, and V. Issarny. COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. *Journal of System and Software*, 80(12), 2007.

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

[56] M.Tivoli and P.Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming*, 71(3):181.

[57] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006.

[58] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *ICDCS*, page 3, 2006.

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

[60] J. Pathak, S. Basu, R. R. Lutz, and V. Honavar. MOSCOE: an Approach for Composing Web Services through Iterative Reformulation of Functional Specifications. *Int. Journal on Artificial Intelligence Tools*, 17(1):109–138, 2008.

[61] P.-G. Raverdy, V. Issarny, R. Chibout, and A. de La Chapelle. A multi-protocol approach to service discovery and access in pervasive environments. In *Proc. of MobiQuitous'06*, pages 1–9. IEEE Computer Society, 2006.

[62] M. Román, R. H. Campbell, and F. Kon. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[63] 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.

[64] G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *SEFM 2008*, page 313.

[65] A. Schmitt and J.-B. Stefani. The kell calculus: A family of higher-order distributed process calculi. In *Global Computing*, pages 146–178, 2004.

[66] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical report, Pittsburgh, PA, USA, 1994.

[67] R. Spalazzese, P. Inverardi, and V. Issarny. A Theory of Mediators for the Ubiquitous Networking Environment - Technical Report TRCS 006/2009 Dipartimento di Informatica, University of L'Aquila, September 2009.

[68] 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.

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

[70] J.-B. Stefani. A calculus of kells. *Electr. Notes Theor. Comput. Sci.*, 85(1), 2003.

[71] M. Stollberg, E. Cimpian, and D. Fensel. Mediating capabilities with deltarelations. In *In Proceedings of the First International Workshop on Mediation in Semantic Web Services, co-located with the Third International Conference on Service Oriented Computing (ICSOC 2005*, 2005.

[72] 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.

[73] J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a Theory of Web Service Choreographies. In *WS-FM'07, LNCS 4937*, page 1.

[74] D. Taubner. Finite representations of ccs and tcsp programs by automata and petri nets. *LNCS 369*, 1989.

[75] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture : foundations, theory, and practice*. Hoboken (N.J.) : Wiley, 2009.

[76] M. Tivoli, P. Fradet, A. Girault, and G. Goessler. Adaptor synthesis for real-time components. In *TACAS 2007, LNCS 4424, Springer-Verlang Berlin Heidelberg*, page 185.

[77] M. Tivoli and P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, 2008.

[78] M. Utting and B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan and Kaufmann, 2006.

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

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

[81] R. J. van Glabbeek. Notes on the methodology of ccs and csp. In *ACP '95: Proceedings from the international workshop on Algebra of communicating processes*, pages 329–349, Amsterdam, The Netherlands, The Netherlands, 1997. Elsevier Science Publishers B. V.

[82] A. Wasylkowski and A. Zeller. Mining Operational Preconditions. http://www.st.cs.uni-saarland.de/models/
papers/wasylkowski-2008-preconditions.pdf (Tech. Rep.).

[83] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *ESEC-FSE '07, pp. 35-44*. ACM, 2007.

[84] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

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

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

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