

## Application-layer Connector Synthesis

Paola Inverardi, Romina Spalazzese, Massimo Tivoli

► To cite this version:

Paola Inverardi, Romina Spalazzese, Massimo Tivoli. Application-layer Connector Synthesis. Marco Bernardo and Valérie Issarny. Formal Methods for Eternal Networked Software Systems (SFM), 6659, Springer-Verlag Berlin Heidelberg, pp.148–190, 2011, LNCS, <10.1007/978-3-642-21455-4\_5>. <inria-00620465>

**HAL Id: inria-00620465**

**<https://hal.inria.fr/inria-00620465>**

Submitted on 8 Sep 2011

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

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

# Application-layer Connector Synthesis<sup>\*</sup>

Paola Inverardi, Romina Spalazzese, and Massimo Tivoli

Dipartimento di Informatica - Università degli Studi dell'Aquila, Italy  
{paola.inverardi,romina.spalazzese,massimo.tivoli}@di.univaq.it

**Abstract.** The heterogeneity characterizing the systems populating the Ubiquitous Computing environment prevents their seamless interoperability. Heterogeneous protocols may be willing to cooperate in order to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other. Despite numerous efforts have been done in the literature, the automated and run-time interoperability is still an open challenge for such environment. We consider interoperability as the ability for two Networked Systems (NSs) to communicate and correctly coordinate to achieve their goal(s).

In this chapter we report the main outcomes of our past and recent research on automatically achieving protocol interoperability via connector synthesis. We consider *application-layer connectors* by referring to two conceptually distinct notions of connector: *coordinator* and *mediator*. The former is used when the NSs to be connected are already able to communicate but they need to be specifically coordinated in order to reach their goal(s). The latter goes a step forward representing a solution for both achieving correct coordination and enabling communication between highly heterogeneous NSs. In the past, most of the works in the literature described efforts to the automatic synthesis of coordinators while, in recent years the focus moved also to the automatic synthesis of mediators. Within the CONNECT project, by considering our past experience on automatic coordinator synthesis as a baseline, we propose a formal theory of mediators and a related method for automatically eliciting a way for the protocols to interoperate. The solution we propose is the *automated synthesis of emerging mediating connectors* (i.e., *mediators* for short).

## 1 Introduction

Today's ubiquitous computing environment is populated by a wide variety of heterogeneous Networked Systems (NSs), dynamically appearing and disappearing, that belong to a multitude of application domains: home automation, consumer electronics, mobile and personal computing, to mention a few. Key technologies such as the Internet, the Web, and the wireless computing devices and networks can be qualified as ubiquitous, in the sense of Mark Weiser [80], even if these technologies have still not reached the maturity envisioned by the Ubiquitous Computing and the subsequent pervasive computing and ambient intelligence

---

<sup>\*</sup> This work has been partially supported by the FET project CONNECT No 231167.

paradigms because of the extreme level of heterogeneity of the underlying infrastructure which prevents seamless interoperability. In this environment, heterogeneous protocols may be willing to cooperate in order to reach some common goal even though they meet dynamically and do not have a priori knowledge of each other.

The term *protocol* refers to *interaction protocols* or *observable protocols*. That is, a protocol is the behavior of a system in terms of the sequences of messages visible at the interface level, which it exchanges with other systems. In this chapter we consider *application-layer* protocols as opposed to *middleware-layer* protocols that are treated in detail in [76].

By referring to the notion of *interoperability* introduced in [30], the problem we address in this chapter, is related to *how to automatically achieve the interoperability between heterogeneous protocols in the Ubiquitous Computing environment*.

With interoperability, we mean the ability of heterogeneous protocols to communicate and correctly coordinate to achieve their goal(s). The communication is expressed as synchronization, i.e., two systems communicate if they are able to synchronize on “common actions”. Coordination is expressed by the achievement of a specified goal, i.e., two systems succeed in coordinating if they interact through synchronization according to the achievement of their goal(s). Communication that is achieved through a complex protocols interaction can be regarded as a simple form of coordination. Indeed, application level protocols introduce a notion of communication that goes beyond single basic synchronizations and may require a well defined sequence of synchronization to be achieved.

In order to make communication and correct (with respect to the specified goal) coordination between heterogeneous protocols possible, we focus on methods, and related tools, for the *automatic application-layer connector synthesis*. In particular, in this chapter, we report our past and recent work on devising automatic connector synthesis techniques in the domains of *Component Based Software Engineering* (CBSE) and *Ubiquitous Computing* (UbiComp), respectively. The work carried on within the CBSE domain can be considered as a baseline for the work done in the UbiComp domain. The latter has been done in the context of the CONNECT project [30] and, with respect to our past work, represents the novel contribution concerning the automatic synthesis of application-layer connectors. However, it is worth mentioning that these two research contributions address two distinct sub-problems of the automatic connector synthesis problem.

In particular, in the CBSE domain, we used automatic connector synthesis in order to face the so-called *component assembly* problem. This problem can be considered as a particular instance of the above mentioned interoperability problem where the issue of enabling communication is assumed to be (almost) already solved. The focus, in the component assembly problem, is on how to coordinate the interactions of already communicating black-box components so that the resulting system is free from possible deadlocks and it satisfies a goal specified in terms of *coordination policies*. Dealing with black-box components, this is done by inserting in the system a software *coordinator*. It is an additional

component beyond the ones forming the system and it is synthesized so as to intercept all component interactions in order to prevent deadlocks and those interactions that violate the specified coordination policies. Coordination policies are routing policies usually specified in some automata-based or temporal logic formalism. Thus, a *coordinator* can be considered as a specific notion of connector, i.e., a *coordination connector*.

Conversely, in the UbiComp domain, the granularity of a system shifts from the granularity of a system of components (as in the CBSE domain) to the one of a *System-of-Systems* (SoS) [27]. An SoS is characterized by an assembly of a wide variety of building blocks. Thus, in the UbiComp domain, enabling communication between heterogeneous NSs regardless, at a first stage, possible coordination mismatches, becomes a primary concern. This introduces another specific notion of connector, i.e., the notion of *mediator* seen as a *communication connector*.

Achieving correct communication and coordination among heterogeneous NSs means achieving interoperability among them. The interoperability problem and the specific notions of connector (e.g., coordinator or mediator) that can be used to solve it, or part of it, have been the focus of extensive studies within different research communities. Protocol interoperability come from the early days of networking and different efforts, both theoretical and practical, have been done to address it in several areas including, for example: protocol conversion, component adaptors, Web services mediation, theories of connectors, wrappers, bridges, and interoperability platforms.

Despite the existence of numerous solutions in the literature, to the best of our knowledge, all of them are more focused on coordinator synthesis and little effort has been devoted to the automatic synthesis of mediators. In particular, these approaches either: (i) assume the communication problem solved (or almost solved) by considering protocols already (or almost) able to interoperate; or (ii) are informal making automatic reasoning impossible; or (iii) follows a semi-automatic process for the mediator synthesis requiring a human intervention; or (iv) consider only few possible mismatches.

Our recent work on mediator synthesis has been devoted in particular to (i) the elicitation and definition of a theory of emerging connectors which also includes related supporting methods and tools. In particular, our recent work has led us to *design automated model-based techniques and tools to support the devised synthesis process*, from protocol abstraction to matching and mapping. Moreover we (ii) characterized protocol mismatches and related mediator patterns, and (iii) we designed a combined approach to take into consideration also non-functional properties while building an interoperability solution. While (i) is part of this chapter, for (ii) and (iii), we refer to [66,65] and [14], respectively.

The remainder of the chapter is organized as follows. Section 2 sets the context of the work reported in this chapter. In particular, by means of two examples, this section clarifies the distinction between the notions of coordinator and mediator. Section 3 describes different approaches for the automatic synthesis of coordinators. Section 4 describes the theory of emerging connectors (i.e., of medi-

ators) mentioned above. Since the pool of coordinator synthesis approaches that are discussed in Section 3 represents the *baseline* chosen from the state-of-the-art in order to devise the approach described in Section 4, for the sake of brevity, the level of description of these two sections is intentionally kept different. That is, Section 3 briefly recalls the different coordinator synthesis approaches by simply providing an overview of them, whereas Section 4 describes in more detail the novel contribution of our recent research with respect to the automatic synthesis of application-layer connectors. Section 5 discusses related works in the areas of both coordinator and mediator synthesis. Section 6 concludes the chapter and outlines our future perspectives in the context of CONNECT.

## 2 Setting the Context

Within the CONNECT project, at synthesis stage, we can assume that a NS comes together with a *Labeled Transition System* (LTS) [42] based specification of its interaction protocol. The interaction protocol of a NS expresses the order in which input and output actions are performed while the NS interacts with its environment. Input actions model methods that can be called, or the end of receiving messages from communication channels, as well as the return values from such calls. Output actions model method calls, message transmission via communication channels, or exceptions that occur during methods execution.

As said in Section 1, our focus is on the automatic synthesis of *application-layer connectors*. Our notion of protocol abstracts from the content of the exchanged data, i.e., values of method/operation parameters, return values, or content of messages. That is, we are interested in harmonizing the behavior protocol (e.g., scheduling of operation calls) of heterogeneous NSs rather than performing mediation of communication primitives or of data encoding/decoding that are issues related to the synthesis of middleware-layer connectors (see the work described in [76]).

As introduced in Section 1, the interoperability problem concerns the problem of both enabling *communication* and achieving correct *coordination*. We recall that in our past research we addressed correct coordination by assuming communication already solved. This is done via automatic coordinator synthesis (Section 3). Instead, our current research focuses on the whole interoperability problem by devising methods and tools for the automatic mediator synthesis (Section 4).

In order to better clarify the distinction between the notions of coordinator and mediator, in the following two sub-sections, we describe two simple yet significant examples of the kinds of interoperability problems that can be solved by using coordinators (Section 2.1) and mediators (Section 2.2).

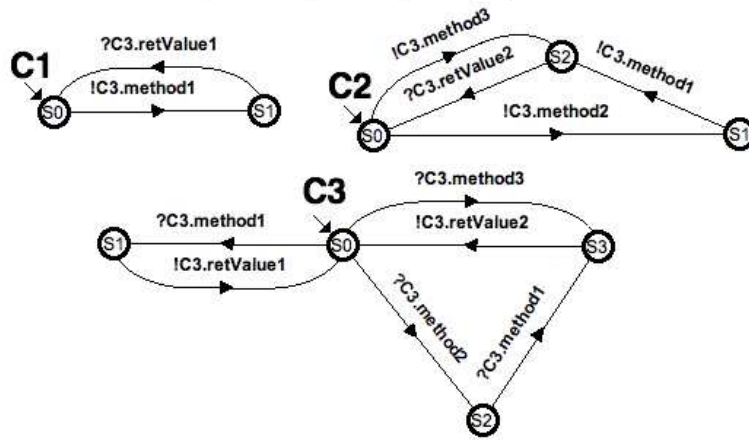
### 2.1 The Need for Coordinators: the Shared Resource Scenario

To better illustrate protocol coordination and the related underlying problems, in the following we describe the Shared Resource scenario. This explanatory example is concerned with the automatic assembly of a client-server component-based

system. This system is formed by three components: two clients, respectively denoted as **C1** and **C2**, and one server denoted as **C3** (the component controlling the Shared Resource). This example, although very simple, exhibits coordination problems that exemplify the kind of problems that coordinator synthesis can solve. For instance, here, the problem is due to the presence of *race conditions* in accessing a shared resource.

Let us assume that we want to assemble a system formed by **C1**, **C2**, and **C3**. In doing so, we want to automatically prevent possible deadlocks and guarantee a specified coordination policy, hence, guaranteeing that the system's goal is reached.

Figure 1 represents the behavior of each component in terms of an LTS.



**Fig. 1.** Components' behavior for the Shared Resource scenario

Each LTS models the component observable behavior in an intuitive way. Each state of an LTS represents a state of the component and the state **S0** represents its initial state. Each action or complementary action performed by interacting with the environment of the component (i.e., all other components in parallel) is represented as a label of a transition into a new state. Actions are input or output. Within an LTS of a component, the label of an input action is prefixed by the question mark “?” (e.g., **?C3.returnValue1** of **C1**). The label of an output action is prefixed by the exclamation mark “!” (e.g., **!C3.method2** of **C2**).

The interface of server **C3** exports three methods denoted as **C3.method1**, **C3.method2**, and **C3.method3**, respectively. While **C3.method2** has no return value, **C3.method1** and **C3.method3** can return some value. **C3.method1** returns two possible return values denoted as **C3.returnValue1**, and **C3.returnValue2**. The former is returned when a call of **C3.method1** has not preceded by a call of

`C3.method2`. Otherwise, the latter is returned. `C3.method3` returns only one value, i.e., `C3.returnValue2`. The two clients perform method calls according to the server interface.

It is worthwhile noticing that the described component interfaces syntactically match since either they already match or suitable component wrappers have been previously developed by the system assembler. As stated above, the problem of enabling communication is here considered as already solved. We recall that, in coordinator synthesis, the focus is on automatically preventing interaction protocol mismatches rather than enabling communication.

By continuing the description of our example, deadlocks can occur because of a race condition among `C1` and `C2`. In fact, one client (i.e., `C2`) performs a call of `C3.method2`, hence leading the server `C3` in a state in which it expects a call of `C3.method1`. While `C2` is attempting to perform the call of `C3.method1`, the other client (i.e., `C1`) performs such a call. In this scenario `C1`, `C2`, and `C3` are in the state `S1`, `S1`, and `S3` of their LTSs. Now, `C3` expects to return `C3.returnValue2` as return value of `C3.method1` but `C2` is still waiting to perform a call of `C3.method1` and `C1` expects a different return value. Thus, a coordination mismatch occurs and it results in an deadlock in the interaction between `C1`, `C2`, and `C3`.

This mismatch can be solved by synthesizing a software coordinator that supervises the components' interaction by preventing the deadlock [73,8,72]. At the level of the coordinator's actual code, the coordinator is synthesized as a multi-threaded component that creates a thread for each request and for each caller performing such a request. Preventing, or solving if possible, deadlocks corresponds to put in a *waiting* state the thread that handles the request leading to the deadlock state and performed by the identified caller. Thus the coordinator will return, again, the control to the caller, for that request, only when it reaches a state in which the blocked request is allowable<sup>1</sup>. Such multi-threaded servers are supported by existing component technologies such as COM/DCOM or CORBA.

Another coordination issue that one can note is that, e.g., `C1` can always obtain the access to the shared resource, while `C2` never obtains it since `C2` can always require the access whenever the resource is already "lock" by `C1`. In other words, `C3` cannot be fair in providing the access to the shared resource it supervises. To solve this issue, a software coordinator can be automatically synthesized so as to enforce an *alternating protocol* policy [73] on the components' interaction. The coordinator allows only the alternating access of `C1` and `C2` to the shared resource.

## 2.2 The Need for Mediators: the Photo Sharing Scenario

To better illustrate protocol mediation and the related underlying problems, in the following we describe the Photo Sharing scenario within a stadium. In

<sup>1</sup> Meaning that, this time, that request performed from that caller does not lead to a deadlock.

general, different versions of the Photo Sharing application may be available on the spectators' handhelds, thus calling for appropriate interoperability solutions.

Let us consider two Photo Sharing implementations: an Infrastructure-based (IB) and an ad hoc peer-to-peer (P2P) respectively shown by Figures 2 and 3. The protocols are depicted using LTSs where the name of actions are self-explanatory. We further use the convention that actions with overbar denote output actions while the ones with no overbar denote input actions.

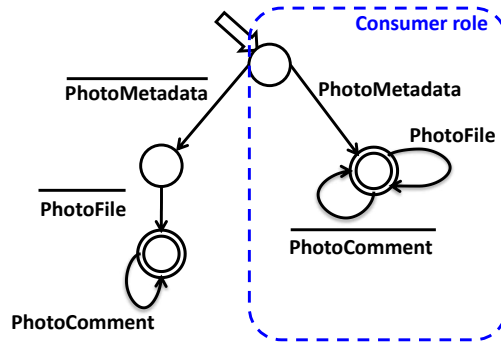


Fig. 2. Peer-to-Peer-based implementation

In the IB implementation, a Photo Sharing service is provided by the stadium, where only authenticated photographers are able to produce pictures while any spectator may download and even annotate pictures.

The P2P implementation allows for photo download, upload and annotation by any spectator, who are then able to directly share pictures using their handhelds.

Then, taking the producer perspective, the high level functionalities that the networked systems implement are: (1) the *authentication* -for the IB producer only- possibly followed by (2) the *upload of photo*, by sending both metadata and file, possibly followed by (3) the *download of comments*; on the other hand, taking the consumer perspective, the implemented high level functionalities are: (i) the *download of photo* by receiving both metadata and file respectively, possibly followed by (ii) the *upload of comments*.

In the P2P implementation, the networked system implements both roles of *producer* and *consumer*. Instead, while having similar roles and high level functionalities, the IB implementation differs with respect to the P2P one because: (i) in IB, the *consumer* and *producer* roles are played by two different/separate networked systems, in collaboration with the server, and (ii) comparing complementary roles among any P2P and IB, they have different interfaces and behaviors.

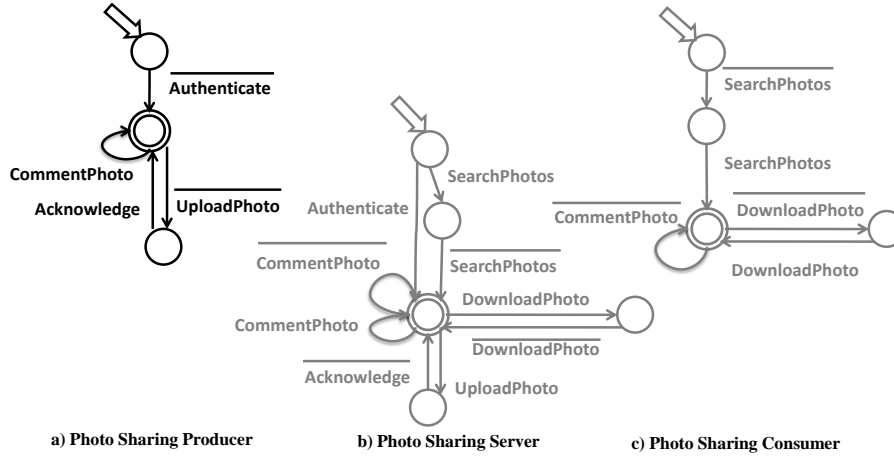


Fig. 3. Infrastructure-based implementation

For the sake of illustration we consider as example the pair of mismatching applications made by: the IB producer (Figure 3 a)) and the P2P Photo Sharing consumer (portion within the dashed line of Figure 2). As can be noticed, the two protocols have different signatures and several discrepancies in the behavior that prevent their direct interoperability. Thus a mediator is needed to solve these heterogeneity in order to enable their communication. A detailed description of the mediator, including the problems it solves, is provided in Section 4.6 where the Photo Sharing scenario is used as running example.

### 3 Automatic Synthesis of Application-layer and Failure-free Coordinators

This section provides an overview of different approaches to the automatic synthesis of application-layer coordinators. We first introduce each approach by outlining their commonalities and differences. Then, through Sections 3.1 to 3.4, we give a complete overview of each approach.

Section 3.1 describes a method for the correct (with respect to coordination mismatches) and automatic assembly of component-based systems via centralized coordinator synthesis [73]. In this context, by considering communication issues already solved, the interoperability problem introduced in Section 1 can be rephrased as follows: *given a set of interacting components,  $C$ , and a set of behavioral properties,  $P$ , automatically derive a deadlock-free assembly,  $A$ , of these components which guarantees every property in  $P$ , if possible.* The assembly  $A$  is a composition of the components in  $C$  plus a synthesized coordinator. The coordinator is synthesized as an additional component which intercepts all the component interactions so as to control the exchange of messages with the aim

of preventing possible deadlocks and those interactions that violate the properties in  $P$ . In [73] this problem is addressed by showing how to automatically synthesize the implementation of a centralized coordinator.

Unfortunately, in a distributed environment it is not always possible or convenient to introduce a centralized coordinator. For example, existing distributed systems might not allow the introduction of an additional component (i.e., the coordinator) which coordinates the information flow in a centralized way. Moreover, the coordination of several components might cause loss of information and bottlenecks hence slowing down the response time of the centralized coordinator. Conversely, building a distributed coordinator might extend the applicability of the approach to large-scale contexts.

To overcome the above limitations, in [8], an extension of the previous method is proposed. This extension is discussed in Section 3.2. The aim of the proposed extension is to automatically synthesize a distributed coordinator into a set of wrappers (local coordinators), one for each component whose interaction has to be controlled. The distributed coordinator synthesis approach has various advantages with respect to the synthesis of centralized coordinators. The most relevant ones are: (i) no centralized point of information flow exists; (ii) the degree of parallelism of the system without the coordinator is maintained; and (iii) all the domain-specific deployment constraints imposed on the centralized coordinator can be removed.

However, both methods are *static*; that is, if the system assembled by means of the synthesized coordinator evolves, e.g., a new component is added, or an existing one is either replaced or removed, the two methods have to be entirely re-performed in order to produce a new coordinator. Since, in the worst case, the computational complexity of the coordinator synthesis is exponential, re-performing the methods whenever a change in the systems occurs cannot be acceptable.

For this reason, in [57], a Software Architecture (SA) based method is proposed in which the usage of the system SA and of SA verification techniques allows the system assembler to design architectural components whose interaction is verified with respect to the specified properties. By exploiting this validation, the system assembler can perform coordinator synthesis by only focusing on each single architectural<sup>2</sup> component, hence refining it as an assembly of actual components which respect the architectural component observable behavior. In this way coordinator synthesis is performed locally on each architectural component, instead of globally on the whole system interactions, hence reducing the state-space explosion phenomenon due to the exponential complexity of the synthesis. The approach can be equally well applied to efficiently manage the whole re-configuration of the system when one or more components need to be updated, still maintaining the required properties. The specified and verified system SA is used as starting point for the derivation of coordinators that are required to apply changes in the composed system. An overview of this approach is given in Section 3.3.

---

<sup>2</sup> It is an ideal component specified in the system SA.

The methods outlined so far have been all applied to real case studies in the domains of COM/DCOM and J2EE applications. This experimentation has been carried on through the SYNTHESIS tool [6] that implements all the outlined methods.

All the previously mentioned methods do not account for the handling of non-functional attributes. Thus, recently, in [72], an extension of SYNTHESIS is proposed for automatically assembling real-time systems. The extended method and related tool, called SYNTHESISRT, are discussed in Section 3.4. This extension accounts for the handling of Quality-of-Service (QoS) attributes such as duration and latency of actions plus component clocks.

### 3.1 Automatic Synthesis of Centralized Application-layer and Failure-free Coordinators

SYNTHESIS is a technique equipped with a tool [6] that permits to assemble a component-based application in a deadlock-free way [73,8]. Starting from a set of components Off The Shelf (OTS), SYNTHESIS assembles them together according to a so called coordinator-based architecture by synthesizing a coordinator that guarantees deadlock-free interactions among components. The code that implements the coordinator is automatically derived directly from the OTS (black-box) components' interfaces. Synthesis assumes a partial knowledge of the components' interaction behavior described as finite state automata plus the knowledge of a specification of the system to be assembled given in terms of Message Sequence Charts (MSCs) [4,74,75]. Furthermore, by exploiting that MSC specification, it is possible to go beyond deadlock. Actually, the MSC specification is an implicit failure specification. That is we assume to specify all the *desired* assembled system behaviors which are failure-free from the point of view of the system assembler, rather than to explicitly specify the failure. Under these hypotheses, SYNTHESIS automatically derives the assembling code of the coordinator for a set of components. The coordinator is derived in such a way to obtain a failure-free system. It is shown that the coordinator-based system is equivalent according to a suitable equivalence relation to the initial one once deparure of all the failure behaviors. The initial coordinator is a *no-op* coordinator that serves to model all the possible component interactions (i.e., the failure-free and the failing ones). Acting on the initial coordinator is enough to automatically prevent both deadlocks and other kinds of failure hence obtaining the failure-free coordinator.

As illustrated in Figure 4, the SYNTHESIS framework realizes a form of system adaptation. The initial software system is changed by inserting a new component, the coordinator, in order to prevent interactions failures.

The framework makes use of the following models and formalisms. An architectural model, the coordinator-based architecture that constrains the way components can interact, by forcing interaction to go through the coordinator. A set of behavioral models for the components that describe each single com-

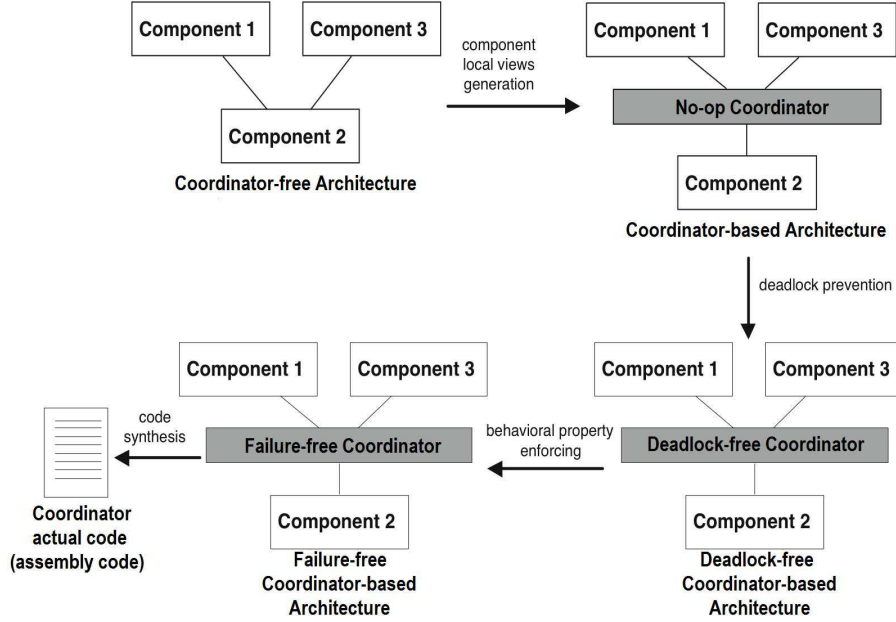


Fig. 4. Automatic synthesis of centralized failure-free coordinators

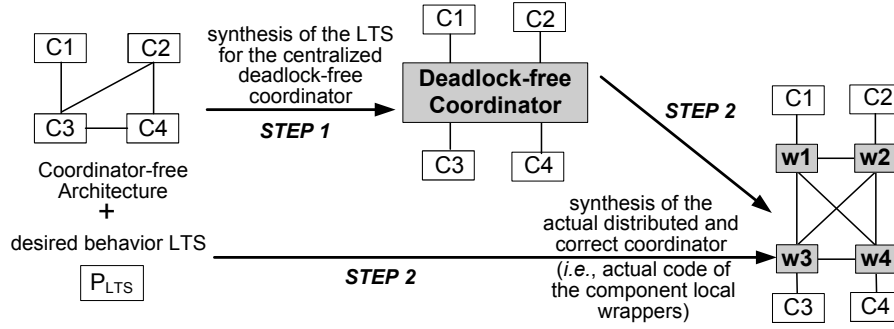
ponent's interaction behavior with the *ideal*<sup>3</sup> external context in the form of LTSs. A behavioral equivalence on LTSs to establish the equivalence among the original system and the adapted/coordinated one. MSCs are used to specify the behavioral integration failure to be avoided, and then LTSs and *LTS synchronous product* [5,42] plus a notion of *behavioral refinement* [49] to synthesize the failure-free coordinator specification, as it is described in detail in [73]. As already mentioned, from the coordinator specification the actual code can then be automatically derived as either a centralized component [73] or a distributed one [8]. The latter is implemented as a set of *wrappers*, one for each component, that cooperatively realize the same behavior as the centralized coordinator. The next section gives an overview of this distributed coordinator synthesis approach.

### 3.2 Automatic Synthesis of Distributed Application-layer and Failure-free Coordinators

As an extension of the method described in Section 3.1, the method that we discuss in this section assumes as input (see Figure 5): (i) a behavioral specification of the coordinator-free system formed by interacting components. It is given as a set  $\{C_1, \dots, C_n\}$  of LTSs (one for each component). The behavior of the system is modeled by composing in parallel all the LTSs and by forcing synchronization

<sup>3</sup> The one expected by the component's developer.

on common events; (ii) the specification of the desired behavior that the system must exhibit. This is given in terms of an LTS, from now on denoted by  $P_{LTS}$ .



**Fig. 5.** Automatic synthesis of distributed failure-free coordinators

These two inputs are then processed in two main steps:

1. by taking into account all component LTSs, we automatically derive the LTS that models the behavior of a centralized deadlock-free coordinator. This first step is inherited from the approach described in Section 3.1 for the synthesis of centralized coordinators. Whenever  $P_{LTS}$  ensures itself deadlock-freeness and its traces are all traces of the centralized coordinator LTS, such a step is not required and, hence, the centralized coordinator cannot be generated. We recall that, at the worst case, the synthesis of the centralized coordinator has an exponential computational complexity in the maximum number of states of the component LTSs. By avoiding the generation of the centralized coordinator, the method's complexity becomes polynomial in the number of states of  $P_{LTS}$ . The first step terminates by checking whether enforcing  $P_{LTS}$  is possible or not. This check is implemented by a suitable notion of refinement. Refinement, in general, formalizes the relation between two LTSs at different level of abstractions. Refinement is usually defined as a variant of *simulation*. In our method, we use a suitable notion of *strong simulation* [49] to check a refinement relation between two LTSs.
2. In the second step, let  $K$  be the LTS of the centralized coordinator. If  $K$  has been generated and it has been checked that  $P_{LTS}$  can be enforced on it, our method explores  $K$  looking for those states representing the *last chance* before entering an execution trace that leads to a deadlock. For instance, in Figure 6, the state S4 represents the last chance state before incurring in the deadlock state S7. This information is crucial for deadlock prevention purposes.

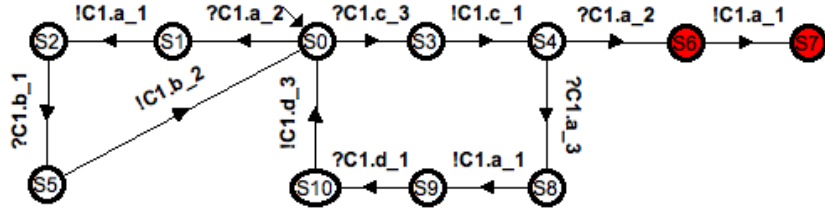


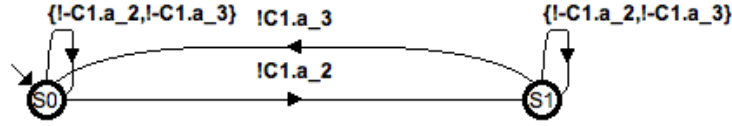
Fig. 6. An example of a centralized coordinator LTS in SYNTHESIS

The search of the last chance states is realized by means of a depth-first search, performed on  $K$ , whose aim is to save those states into the local wrappers of the components that could lead the system from a last chance state to a deadlock by means of a so called *critical action*. The idea is therefore not to allow a component to perform a critical action before being sure that the system will not reach a deadlock state. By interacting with the SYNTHESIS tool, the user can tag component actions as either *controllable* or *uncontrollable* by the external environment. If such a critical action is controllable then it can be discarded. Otherwise, if it is uncontrollable, SYNTHESIS performs a *controller synthesis step* [60,16] that “backtracks” by looking for the first controllable action that can be discarded to prevent the execution of the critical action. After the execution of this depth-first search on  $K$ , the set of last chance states and associated critical actions are stored in a table, one for each component wrapper.

The second step also explores  $P_{LTS}$  to retrieve information crucial for undesired behavior prevention. The aim here is to split and distribute  $P_{LTS}$  in a way that each local wrapper knows which actions the wrapped component is *allowed* to execute. This is realized by means of a depth-first search on  $P_{LTS}$ .

Referring to Figures 6 and 7 for instance, the wrapper of component  $C3$  must not allow the component to send the request  $C1.a$ , if the current global state of the system matches the state  $S0$  in  $P_{LTS}$ , hence enforcing the desired behavior modeled by  $P_{LTS}$ . In particular, the label  $\{!-C1.a_2, !-C1.a_3\}$  of the loop on  $S0$  denotes two loops, one labeled with  $!-C1.a_2$  and one labeled with  $!-C1.a_3$ . The action  $!C1.a_3$  denotes an output action  $C1.a$  by  $C3$ ;  $!-C1.a_3$  represents *its negation*, i.e., all possible actions different from it.

The sets of *last chance states* and *allowed actions* are stored and, subsequently, used by the local wrappers as basis for correctly synchronizing with each other by exchanging additional communication. In other words, the local wrappers interact with each other to restrict the components’ standard communication (modeled by  $K$ ) by allowing only the part of the communication that is correct with respect to deadlock-freeness and  $P_{LTS}$ . By de-



**Fig. 7.** An example of a desired behavior LTS in SYNTHESIS

centralizing  $K$ , the local wrappers preserve parallelism of the components forming the system.

The message exchange among wrappers for synchronization purposes is realized by means of the two procedures *Ask* and *Ack*, whose implementation is automatically synthesized by SYNTHESIS. The first is used to ask the permission to the other wrappers before allowing a component to proceed with a critical action. The second is used to reply to a message sent by procedure *Ask* when the global state is safe.

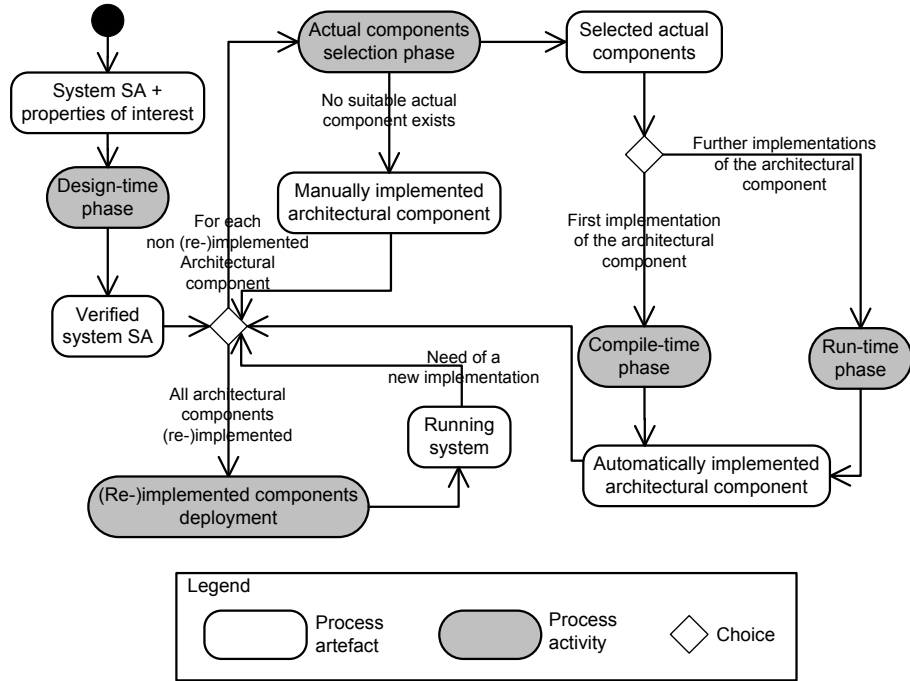
### 3.3 Automatic Synthesis of Application-layer Coordinators for Evolvable Systems

This coordinator synthesis method is composed of four main phases organized as shown in Figure 8. In the following, the description of the method assumes that an SA has been modeled by using the CHARMY framework [56] (*System SA + properties of interest* in Figure 8).

**Design-time phase:** the first phase concerns the system SA verification. This phase is performed by using CHARMY. The input of this phase is an SA and the properties that one wants to check. The output is a system SA specification that respects the properties of interest (*Verified system SA* in Figure 8).

For each verified architectural component that has not yet been implemented, the *Actual components selection phase* is performed. After that all the architectural components have been implemented, they are deployed (*Re-implemented components deployment* in Figure 8) hence producing a first running version of the system (*Running system* in Figure 8).

**Actual components selection phase:** our method implements each architectural component as an assembly of actual components acquired from a third-party, when possible. This phase aims at selecting third-party components by looking at their interfaces and functionalities. For the selection criteria used to establish which actual components have to be acquired to implement an architectural component, we refer to [57] where the method is discussed in detail. This phase takes as input a verified architectural component and it is performed with respect to a repository of actual components acquired from a third-party [54] (black-box components). The output is the set of actual components selected as possible candidates for the implementation of the architectural one or an empty



**Fig. 8.** Automatic synthesis of failure-free coordinators for evolvable systems

set. In case of an empty set, the architectural component is manually implemented (*Manually implemented architectural component* in Figure 8) since we did not find suitable components that can be assembled to implement the considered architectural component. In this case it is up to the developer to guarantee that the component implementation conforms to its architectural specification, e.g., via verification techniques.

If possible candidates are found (*Selected actual components* in Figure 8) they could still need some adaptations (e.g., they might provide more functionalities as needed or interaction mismatches might occur). The compile-time phase and the run-time phase will automatically manage that in the first implementation of the architectural component and in its further implementations, respectively.

**Compile-time phase:** in order to correctly implement the considered architectural component, this phase automatically produces an assembly of the selected actual components that is correct with respect to the architectural component’s observable behavior (*Automatically implemented architectural component* in Figure 8). This is done by exploiting the SYNTHESIS tool as either described in Sections 3.1 or 3.2 depending on which kind of implementation is required for the architectural component, centralized or distributed.

**Run-time phase:** when a new implementation of an architectural component is needed (the transition *Need of a new implementation* outgoing from *Running system*), the correct (re-)implementation of the considered architectural component is produced analogously to what is done in the compile-time phase, i.e., again via the SYNTHESIS tool. The run-time phase performs additional operations with respect to the compile-time phase. These operations are the suspension of the running system in a consistent state and the transfer of the computational state.

### 3.4 Automatic Synthesis of Application-layer Coordinators for Real-time Systems

Recently, the SYNTHESIS approach and its related tool has been extended to the context of real-time systems [72]. This extension, hereafter called SYNTHESISRT, has been developed by the Software Engineering research group at University of L'Aquila in cooperation with the POP ART project team at INRIA Rhône-Alpes. In [72], it is shown how to deal with the compatibility, communication, and QoS issues that can raise while building a real-time system from reusable black-box components within a lightweight component model where components follow a data-flow interaction model. Each component declares input and output ports which are the points of interaction with other components and/or the execution environment. Input (resp., output) ports of a component are connected to output (resp., input) ports of a different component through synchronous links. Analogously to the version of SYNTHESIS without real-time constraints, a component interface includes a formal description of the *interaction protocol* of the component with its expected environment in terms of sequences of writing and reading actions to and from ports. The interface language is expressive enough to specify QoS constraints such as writing and reading *latency*, *duration*, and *controllability*, as well as the component's *clock* (i.e., its activation frequency). In order to deal with incompatible components (e.g., clock inconsistency, read/write latency/duration inconsistency, mismatching interaction protocols, etc.) we synthesize coordinators interposed between two or more interacting components. A coordinator is a component that mediates the interaction between the components it supervises, in order to harmonize their communication. Each coordinator is automatically derived by taking into account the interface specification of the components it supervises. The coordinator synthesis allows the developer to automatically and *incrementally* build *correct-by-construction* systems from third-party components.

Figure 9 shows the main steps of the method performed by SYNTHESISRT by also highlighting the used formalisms/models.

We take as input the architectural specification of the network of components to be composed and the component interface specifications. The behavioral models of the components are generated in form of LTSs that make the elapsing of time explicit (step 1). Connected ports with different names are renamed such that complementary actions have the same label in the component LTSs (see

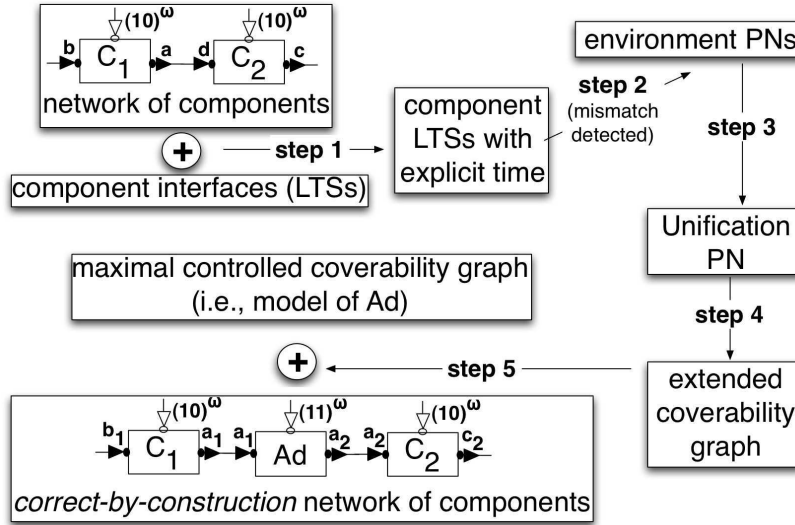


Fig. 9. Main steps of the coordinator synthesis for real-time components

actions  $a$  and  $d$  in Figure 9). Possible mismatches/deadlocks are checked by looking for possible sink states into the parallel composition of the LTSSs. The coordinator synthesis process starts only if such deadlocks are detected.

The synthesis first proceeds by constructing a *Petri net* (PN) [52] representation of the environment expected from a component in order not to block it (step 2). It consists in complementing the actions in the component LTSSs that are performed on connected ports, considering the actions performed on unconnected ports as internal actions. Moreover, a buffer storing read and written values is modeled as a place in the environment PN for each IO action. Each such PN represents a partial view of the coordinator to be built. It is partial since it reflects the expectation of a single component. In particular, a write (resp. read) action gives rise to a place (buffer) without outgoing (resp. incoming) arcs.

The partial views of the coordinator are composed together by building causal dependencies between the reading/writing actions and by unifying time-elapsing transitions (step 3). Furthermore, the places representing the same buffer are merged in one single place. This *Unification PN* models a coordinator that solves deadlocks using buffers to desynchronize received events from their emission.

However, the unification PN may be not completely correct, in the sense that it can represent a coordinator that may deadlock and/or that may require unbounded buffers. In order to obtain the most permissive and correct coordinator, we generate an extended version of the graph usually known in PNs theory [52] as the coverability graph [29] (step 4).

Our method automatically restricts the behavior of the coordinator modeled by the extended coverability graph in order to keep only the interactions that

are deadlock-free and that use finite buffers (i.e., bounded interactions). This is done by automatically constructing, if possible, an “instrumented” version of our extended coverability graph, called the *Controlled Coverability Graph (CCG)*. The CCG is obtained by pruning from the extended coverability graph both the *sinking* paths and the *unbounded* paths, by using a *controller synthesis* step [61] (step 5). *Ad*, in the figure, denotes the synthesized coordinator.

This process also performs a *backwards error propagation* step in order to correctly take into account the case of sinking and unbounded paths originating from the firing of uncontrollable transitions.

If it exists, the maximal CCG generated is the LTS modeling the behavior of the correct (i.e., deadlock-free and bounded) coordinator. This coordinator models the correct-by-construction assembly code for the components in the specified network. If it does not exist, a correct coordinator assembling the components given as input to our method cannot be automatically derived, and hence our method does not provide any assembly/coordination code for those components.

## 4 Automatic Synthesis of Application-layer Mediators

This section describes our recent work on the automatic synthesis of application-layer mediators. We overview our methodology in Section 4.1 and we give formal foundations in Section 4.2. Then we provide the formalization of our theory by respectively presenting the protocol abstraction in Section 4.3, protocol matching in Section 4.4, and protocol mapping in Section 4.5. Finally we illustrate the application of the theory to the Photo Sharing scenario in Section 4.6. Abstraction, matching, and mapping are fundamental operations of our mediator synthesis approach.

As already illustrated in Section 1, we focus on the interoperability problem between heterogeneous protocols within the UbiComp environment. For the sake of simplicity, and without loss of generality, we limit the number of protocols to two but the work can be generalized to an arbitrary number of protocols.

In particular, we focus on **compatible** or **functionally matching protocols**. Functional matching means that heterogeneous protocols can *potentially communicate* by performing *complementary sequences of actions* (or *complementary conversations*).

*Potentially* means that communication may not be achieved because of *mismatches* (heterogeneity), i.e., the languages of the two protocols are different, although semantically equivalent. For example, protocol languages can have: (i) different granularity, or (ii) different alphabets. Protocols behavior may have, for example, different sequences of actions because of (a.1) the order in which actions are performed by a protocol is different from the order in which the other protocol performs the same actions; (a.2) interleaved actions related to *third parties communications* i.e., other systems, the environment. In some cases, as for example (i), (ii) and (a.1), it is necessary to properly perform a manipulation of the two languages. In the case (a.2) it is necessary to provide an abstraction of

the two actions sequences that results in sequences containing only actions that are relevant to the communication.

*Communication* is then possible if the two possibly manipulated (e.g., re-ordered) and abstracted sequences of actions are complementary, i.e., are the same sequences of actions while having opposite output/input “type” for all actions.

Therefore, the problem we address and overcome, is the *interoperability between heterogeneous protocols in the UbiComp environment*.

With **interoperability**, we mean the property referring to the ability of heterogeneous protocols *to communicate and coordinate* to reach their goal(s). Communication and coordination are expressed by synchronization, i.e., two systems succeed in coordinating if they are able to synchronize hence reaching their goal(s).

In order to make communication between heterogeneous protocols possible, we proposed as **solution** a *theory of mediators* [36,67,64] that we revise and extend in the remainder of this section. The theory, reasoning about the mismatches of the compatible protocols, automatically elicits and synthesizes an *emerging mediator* that solves them allowing protocol interoperability. The theory paves the way for run-time (or on-the-fly) approaches to the mediators synthesis.

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

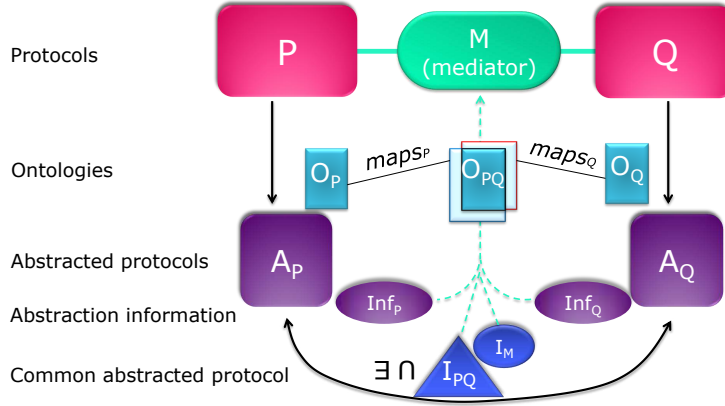
We *assume* that each device, e.g. PDA, smartphone, or tablet, is equipped, for its applications, with the (i) behavioral specification and their (ii) semantical characterization of their actions through ontologies. Taking the perspective of two systems that have compatible protocols and that also communicate with third parties, we assume that there exists also (iii) the proper environment for them, i.e., the other systems representing third parties. Further, we concentrate on application layer interoperability while assuming solved the heterogeneity of the underlying layers.

#### 4.1 Towards Emerging Mediators

Figure 10 depicts the main elements of our methodology which we describe in the following.

The method includes:

- (i) Two application-layer protocols  $P$  and  $Q$  whose representation is given in terms of LTSs, where the *initial* and *final states* on the LTSs define the *sequences of actions* (traces) that characterize the *coordination policies* of the protocols.
- (ii) Two *ontologies*  $O_P$  and  $O_Q$  describing the meaning of  $P$  and  $Q$ ’s actions, respectively.
- (iii) Two *ontology mapping functions*  $maps_P$  and  $maps_Q$  defined from  $O_P$  and from  $O_Q$  to a common ontology. The intersection  $O_{PQ}$  on the common ontology identifies the “common language” between  $P$  and  $Q$ . For simplicity, and



**Fig. 10.** An overview of our approach

without loss of generality, we consider protocols  $P$  and  $Q$  that have disjoint languages and that are minimal where we recall that every finite LTS has a unique minimal representative LTS.

- (iv) Then, starting from  $P$  and  $Q$ , and based on the ontology mapping, we build two abstractions  $A_P$  and  $A_Q$  by relabeling  $P$  and  $Q$ , respectively, where the actions not belonging to the common language  $O_{PQ}$  are hidden by means of silent actions ( $\tau$ s); moreover, we store some abstraction information (i.e., used to make the abstraction),  $Inf_P$  and  $Inf_Q$ , that in case of positive matching check, will be exploited to synthesize the mediator during the mapping;
- (v) Then, we check the compatibility of the protocols by looking for complementary traces (the set  $I_{PQ}$  in figure), modulo mismatches and third parties communications, between the sets of traces  $T_P$  and  $T_Q$  generated by  $A_P$  and  $A_Q$ , respectively. If this is the case, then we are able to synthesize a mediator that makes it possible for the protocols to coordinate. Hence, we store the matching information (i.e., used to make the abstraction)  $I_M$  that will be exploited during the mapping.
- (vi) Finally, given two protocols  $P$  and  $Q$ , and an environment  $E$ , the mediator  $M$  that we synthesize is such that when building the parallel composition  $P||Q||E||M$ ,  $P$  and  $Q$  are able to coordinate by reaching their final states under the hypothesis of fairness.

## 4.2 Formal Foundations

The application-layer interaction protocol, as described in Section 1, is the behavior of a system in terms of the actions it exchanges with other application-

layer interaction protocols. In this section, a characterization of such protocols is provided together with a conceptualization of the application actions.

### Protocols as LTS

As mentioned in Section 2, we use LTSs to characterize the protocols. LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioral languages such as process algebras. Let  $Act$  be the set of observable actions (input/output actions), we get the following definition for LTS:

**Definition 1 (LTS).** *A LTS  $P$  is a quadruple  $(S, L, D, s_0)$  where:*

*$S$  is a finite set of states;*

*$L \subseteq Act \cup \{\tau\}$  is a finite set of labels (that denote observable actions) called the alphabet of  $P$ .  $\tau$  is the silent action. Labels with an overbar in  $L$  denote output actions while the ones without overbar denote input actions. We also use the convention that for all  $l \in L, \bar{\bar{l}} = l^4$ .*

*$D \subseteq S \times L \times S$  is a transition relation;*

*$s_0 \in S$  is the initial state.*

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

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

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

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

**Definition 2 (Trace or Coordination Policy).** *Let  $P = (S, L, D, F, s_0)$ .*

*A trace  $t = l_1, l_2, \dots, l_n \in L^*$  is such that:*

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

We use the usual compact notation  $s_0 \xRightarrow{t} s_n$  to denote a trace, where  $t$  is the concatenation of actions of the trace.

Moreover we define a **subtrace** as any sequence in a protocol (it may be also a trace). More formally:

<sup>4</sup> We inherit this convention from *Calculus of Communicating Systems* (CCS) [49]

**Definition 3 (Subtrace).** Let  $P = (S, L, D, F, s_0)$ .

A subtrace  $st = l_i, l_{i+1}, \dots, l_n \in L^*$  is such that:

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

Similarly to traces, also in this case we use the compact notation  $s_i \xRightarrow{st} s_n$ .

LTSs can be combined using the LTS parallel composition operator. Several semantics have been given in the literature for this operator. The one needed here is similar to the one of CSP (*Communicating Sequential Processes*) [63]: protocols  $P$  and  $Q$  synchronize on complementary actions while proceeding independently when engaged in non complementary actions. Moreover, we need a *synchronous* reference model as the one of CSP or FSP (*Finite State Process*) [47] where the synchronization is forced when an interaction is possible. Differently, the asynchronous model like the one of CCS [49], would allow agents to non-deterministically choose to not interact by performing complementary actions  $a$  and  $\bar{a}$  separately.

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

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

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

$$\frac{P \xrightarrow{m} P'}{P||Q \xrightarrow{m} P'||Q} \quad (\text{where } m \in L_P \wedge \bar{m} \notin L_Q)$$

$$\frac{Q \xrightarrow{m} Q'}{P||Q \xrightarrow{m} P||Q'} \quad (\text{where } m \in L_Q \wedge \bar{m} \notin L_P)$$

$$\frac{P \xrightarrow{m} P'; Q \xrightarrow{\bar{m}} Q'}{P||Q \xrightarrow{\tau} P'||Q'} \quad (\text{where } m \in L_P \wedge \bar{m} \in L_Q)$$

Note that when we build the parallel composition of protocols  $P$ ,  $Q$ , with the environment  $E$ , and the mediator  $M$ , the composed protocol  $P||Q||E||M$  is restricted to the language made by the union of the common languages between

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

### Ontologies

Ontologies play an important role in realizing connectors which primarily relies on reasoning about systems functionalities. More in detail, what is needed is to identify matching sequences of observable actions among the actions performed by the systems. Ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of NSs.

In the literature, [40,39] the ontologies and the ontology mapping are defined as follows:

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

Towards enabling mediators, in the next section we will detail application ontologies characterizing the application actions.

### 4.3 Abstraction Formalization

Given the definition of extended LTS associated with two interaction protocols run by NSs, we want to identify whether such two protocols are *functionally matching* and, if so, to synthesize the mediator that enables them to interoperate, despite behavioral mismatches and third parties communications.

We recall that with *functional matching*, we mean that given two systems with respective interaction protocols  $P$  and  $Q$ , ontologies  $O_P$  and  $O_Q$  describing their actions, ontology mapping functions  $maps_P$  on  $P$  and  $maps_Q$  on  $Q$ , and their intersecting common ontology  $O_{PQ}$ , there exists *at least one pair of complementary traces* (with one trace in  $P$  and one in  $Q$ ) that allows  $P$  and  $Q$  to coordinate. In other words, one or more sequences of actions of one protocol can synchronize with one or more sequences of actions in the other. This can happen by properly solving mismatches, using the basic patterns discussed in [66,65], and managing communications with third parties. Thus, we expect to find, at a given level of abstraction, a common protocol  $C$  that represents the potential interactions of  $P$  and  $Q$ . This leads us to formally analyze such alike protocols to find - if it exists -  $C$  and a suitable mediator that allows the interoperability that otherwise would not be possible. This problem can be formulated as a kind of anti-unification problem [35,58,79,62].

In order to find the protocols’ abstractions, we exploit the information contained in the ontology mapping to suitably relabel the protocols. Specifically,

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

**Fig. 11.** Ontology mapping between Infrastructure-based Photo Sharing Producer and peer-to-peer Photo Sharing (Figure 3 a) and Figure 2 respectively)

as detailed in the following, the relabeling of LTSs produces new LTSs that are labeled only by common actions and  $\tau$ s, and hence are more abstract than before (e.g., sequences of actions may have been compressed into single actions). For illustration, Figure 11 summarizes the ontological information of the IB Producer of Figure 3 a) (first column) and of the P2P Photo Sharing of Figure 2 (third column). The second column shows their *common language*. We recall that: (1) the overlined actions are output/send action while non-overlined are input/receive; (2) the P2P application implements both roles, producer and consumer, while the IB application we are focusing on, is the producer role only (the overall Photo Sharing is implemented by three separate IB applications). This explains why we have in the table two non-paired actions; because they are paired with the actions of the other IB applications.

In the following we describe more formally the abstraction step. We specialize the definition of total ontology mapping of Section 4.2, that maps single elements of  $S_1$  into single elements of  $S_2$ , by defining an *abstraction ontology mapping* that maps the  $S_1$  language (i.e.,  $S_1^*$ ) into  $S_2$ , i.e.,  $maps : S_1^* \rightarrow S_2$ .

We use such specialized ontology mapping on the ontologies of the compatible protocols, where the vocabulary of the source ontology is the language of the protocol. More formally:

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$ ,
- $O_P = (L_P^*, A_P)$  be the ontology of  $P$ ,
- $O = (L, A)$  be an ontology referred as abstract ontology,
- $st \in L_P^*$  be a subtrace on  $P$ .

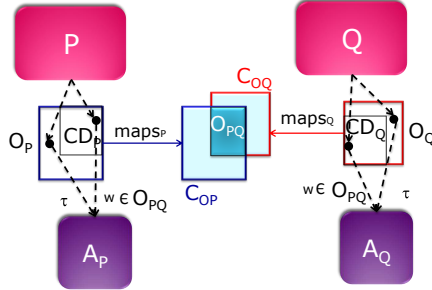
*The abstraction ontology mapping is a function maps such that:*  
 $maps : L_P^* \rightarrow L$ .

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

The abstract protocols are then obtained, leveraging on the abstraction ontology mapping, by relabeling protocols with labels of their common language and  $\tau$ s for the thirds parties languages. A necessary condition for the existence of a common language between two protocols  $P$  and  $Q$ , is that there exist two abstraction ontology mapping  $maps_P$  on  $P$  and  $maps_Q$  on  $Q$  that map the languages  $L_P^*$  of  $P$  and  $L_Q^*$  of  $Q$  into the same/common abstract ontology. Thus, to identify the common language, we first map each protocol's ontology into a common ontology and then by intersection, we find their common language.

Operationally, we do not work on protocols while we reason on traces: starting from a protocol  $P$  ( $Q$  resp.), we extract all the traces from it and apply the relabelling on the traces that result into a set of abstracted traces, with labels belonging to the common language and  $\tau$ s. However, the abstract protocol(s) of  $P$  ( $Q$  resp.), can be easily obtained by merging the traces where possible (e.g. common prefixes). Similarly, we use a reasoning on traces also for the matching and mapping phases.

It has to be noticed that the set of all the traces may not be finite. Then, the abstraction ontology mapping can be applied to an infinite set of traces. We consider minimal protocols<sup>5</sup>. Hence, the infinite set of traces is represented by a minimal automaton (containing at least a final state). Then, the abstraction ontology mapping on such minimal automaton, either applies directly (to the minimal automaton) returning a set of (abstracted) traces on the common language and  $\tau$ s, or it does not exist any automaton unfolding on which the abstraction ontology mapping applies.



**Fig. 12.** The abstract protocol building

Figure 12 depicts the abstraction of the protocols. Let us consider two minimal and deterministic protocols  $P$  and  $Q$  with their respective ontologies

<sup>5</sup> This is similar to the normal form of a system of recursive equations in [37] which is based on the idea to eliminate repetitions of equivalent recursive equations (that is equations with the same unfolding).

$O_P = (L_P^*, A_P)$  and  $O_Q = (L_Q^*, A_Q)$  and their abstraction ontology mappings  $maps_P$  and  $maps_Q$  respectively.

We first map  $O_P$  and  $O_Q$ , through  $maps_P : L_P^* \rightarrow L$  and  $maps_Q : L_Q^* \rightarrow L$  respectively, into a common ontology  $O = (L, A)$  where  $C_{OP}$  and  $C_{OQ}$  represent the codomain sets of  $maps_P$  and  $maps_Q$  respectively.

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

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

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

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

$$O_{PQ} = \{l : l \text{ (or } \bar{l}) = maps_P(st_P) \wedge l \text{ (or } \bar{l}) = maps_Q(st_Q) \}$$

where  $st_P, st_Q$  implement basic mismatches (as defined in papers [66,65]).

For instance, the pairs of labels  $(\overline{UP}, UP)$ , or  $(UP, UP)$ , or  $(UP, \overline{UP})$ , or  $(\overline{UP}, \overline{UP})$  let us derive  $UP$  as an action belonging to the common language.

The abstract protocol  $A_P$  ( $A_Q$  resp.) of  $P$  ( $Q$  resp.), is built as follows:

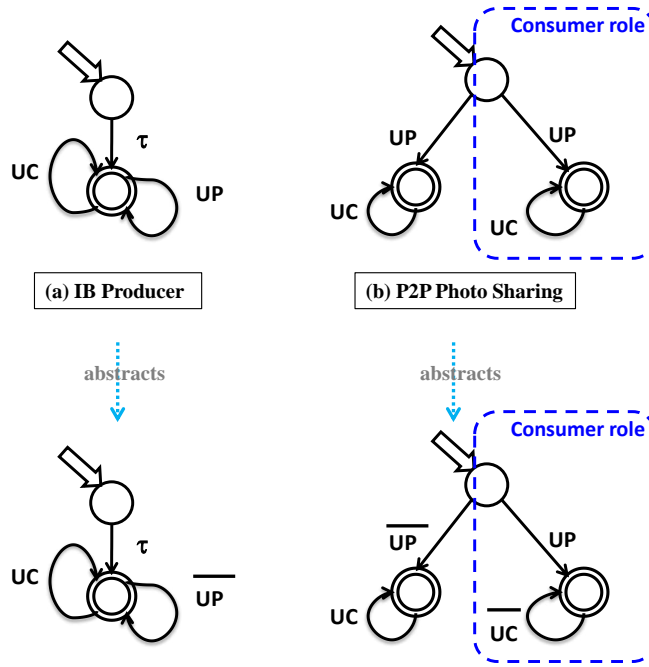
for each trace  $t_P$  of  $P$  ( $t_Q$  of  $Q$  resp.) build a new trace  $t'_P$  ( $t'_Q$ ) such that:

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

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

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$  and  $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$  be protocols,
- $O_P = (L_P^*, AX_P)$  and  $O_Q = (L_Q^*, AX_Q)$  be ontologies of  $P$  and  $Q$  respectively,
- $O = (L, A)$  be a common ontology for  $P$  and  $Q$ ,



**Fig. 13.** Abstracted LTSs of the Photo Sharing protocols

- $maps_P : L_P^* \rightarrow L$  and  $maps_Q : L_Q^* \rightarrow L$  be abstraction ontology mappings of  $P$  and  $Q$  respectively,
- $C_{OP}$  and  $C_{OQ}$  be the codomain sets of  $maps_P$  and  $maps_Q$  respectively,
- $O_{PQ}$  be the common language between  $P$  and  $Q$ .

The relabeling function *relabels* is defined as:  $relabels : (P, maps_P, O_{PQ}) \rightarrow A_P$  where  $A_P = (S_A, L_A, D_A, F_A, s_{0_A})$  and where

$$S_A \subseteq S_P,$$

$$L_A = \{l \in O_{PQ}\} \cup \{\tau\},$$

$$D_A = \{s_i \xrightarrow{l} s_j \text{ (or } s_i \xrightarrow{\bar{l}} s_j) : \exists s_k \xrightarrow{w} s_n \in D_P \wedge l \text{ (or } \bar{l}) = maps_P(w)\},$$

$$F_A \subseteq F_P, \text{ and}$$

$$s_{0_A} = s_{0_P}.$$

The above definition applies similarly to  $Q$ :  $relabels : (Q, maps_Q, O_{PQ}) \rightarrow A_Q$ .

In the Photo Sharing scenario, the only label that is not abstracted in the common language is *authenticate* that represents a third party coordination. The IB producer and P2P Photo-Sharing version 1's abstracted LTSs are shown in Figure 13 where the upper part illustrates the protocols on the common language (i.e., common labels without taking into account output and input) while the bottom part of the figure illustrates the protocols on the common language projected on the protocols (i.e., labels where output and inputs are not abstracted).

The subsequent step is to check whether the two abstracted protocols share a *complementary coordination policy*, i.e., whether the abstracted protocols may in fact synchronize, which we check over protocol traces as mentioned before.

#### 4.4 Matching Formalization

The formalization described so far is needed to: (1) characterize the protocols and (2) abstract them into protocols on the same alphabet. Then, to establish whether two protocols  $P$  and  $Q$  can interoperate given their respective abstractions  $A_P$  and  $A_Q$  based on their common ontology  $O_{PQ}$  (i.e., common language) and possibly  $\tau$ s, we need to check that the abstracted protocols  $A_P$  and  $A_Q$  share complementary coordination policies. To establish this, we use the *functional matching relation* between  $A_P$  and  $A_Q$ , which succeeds if  $A_P$  and  $A_Q$  have a *set of pairs of complementary coordination traces*, i.e., at least one pair.

Before going into the definition of the compatibility or functional matching relation, let us provide the one of *complementary coordination policies*. Informally, two coordination policies are complementary if and only if they are the same sequence of actions while having opposite input/output type for all actions. That is, traces  $t$  and  $t'$  are complementary if and only if: each output action (resp. input) of  $t$  has its complementary input action (resp. output) in  $t'$  and similarly with switched roles among  $t'$  and  $t$ . More formally:

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

- $P = (S_P, L_P, D_P, F_P, s_{0_P})$  and  $Q = (S_Q, L_Q, D_Q, F_Q, s_{0_Q})$ ,
- $A_P, A_Q$  be the abstracted protocols of  $P$  and  $Q$  respectively,
- $T_P$  and  $T_Q$  be the set of all the traces of  $A_P$  and  $A_Q$ , respectively,
- $t = l_1, l_2, \dots, l_n \in T_P$  and  $t' = l'_1, l'_2, \dots, l'_m \in T_Q$ .

*Coordination policies  $t$  and  $t'$  are complementary coordination policies iff the following conditions hold: discarding the  $\tau$ s,*

- (i) *for each  $l_i \in t : l_i$  is an output action (input action resp.)  $\exists l'_j \in t' : l'_j$  is an input action (output action resp.);*
- (ii) *for each  $l'_j \in t' : l'_j$  is an output action (input action resp.)  $\exists l_i \in t : l_i$  is an input action (output action resp.);*

Note that (i) and (ii) above do not take into account the order in which the complementary labels  $l_i$  and  $l'_j$  are within the traces. Hence, two traces having all complementary labels (skipping the  $\tau$ s) but in different order are considered to be complementary coordination policies (modulo a reordering). Therefore, while doing this check, we store such information that will be used during the mediator synthesis in addition to other information, e.g., the abstraction information.

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

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

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

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

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

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

#### 4.5 Mapping Formalization

Given two protocols  $P$  and  $Q$  that functionally match, where the set  $C$  is made by their pairs of complementary coordination policies, we want to synthesize a mediator  $M$  such that the parallel composition  $P||M||Q$ , allows  $P$  and  $Q$  to evolve, for their portion  $C$ , to their final states. An action of  $P$  and  $Q$  can belong either to the *common language* or the *third parties language*, i.e., the environment. We build the mediator in such a way that it lets  $P$  and  $Q$  evolve independently for the portion of the behavior to be exchanged with the environment (denoted by  $\tau$  action in the abstracted protocols) until they reach a “synchronization state” from which they can synchronize on complementary actions. We recall that the synchronization cannot be direct since the mediator needs to perform suitable manipulations as for instance actions reordering or translation according to the ontology mapping. An example of translation in the Photo Sharing scenario is  $UC = CommentPhoto$  in one protocol and  $\overline{UC} = PhotoComment$  in the other.

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

Therefore, a mediator is a protocol that, for each pair  $c_{ij} = (c_i, c_j)$  in  $C$ , builds a mediating trace  $m_{ij}$  such that, for each action (also  $\tau$ ) in  $c_i$  and in  $c_j$  it always first receive the action and then properly resend it. More formally:

**Definition 10 (*Mediator*).** *Let:*

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

The mediator  $M$  for  $C$  is defined as follows:

$$\begin{aligned} & \forall (c_i, c_j) \exists \text{ a mediating trace } m_{ij} \in M : m_{ij} = l_1, l_2, \dots, l_k \wedge k = n + m \wedge \\ & \text{if } l_n = \bar{a} \wedge a \in O_C \wedge a \in c_i \text{ then } \exists 1 \leq h < n : l_h = a \wedge a \in c_j; \\ & \text{if } l_n = \bar{a} \wedge a \in O_C \wedge a \in c_j \text{ then } \exists 1 \leq h < n : l_h = a \wedge a \in c_i; \end{aligned}$$

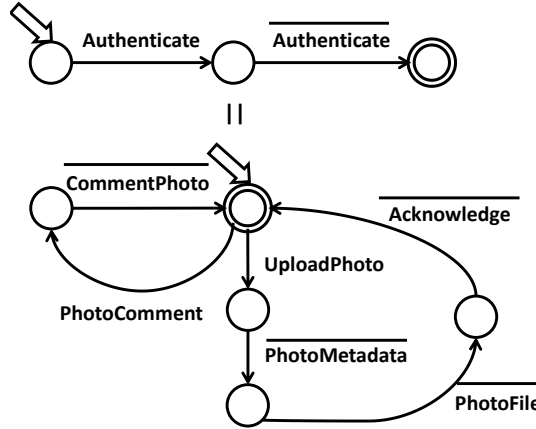
The mediator is logically made up of two separate components:  $M_C$  and  $M_T$ .  $M_C$  speaks only the common language and  $M_T$  speaks only the third parties language.  $M_C$  is a LTS built starting from the common language between  $P$  and  $Q$  whose aim is to solve the protocol-level mismatches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

#### 4.6 Application of the Theory to the Scenario

As already mentioned in Section 4, we assume to have the behavioral specification of the considered Photo Sharing applications, their respective ontologies describing their actions, and the abstraction ontology mapping that defines the common language between IB producer and P2P Photo Sharing. The first step is to *abstract* the protocols exploiting the ontology mapping. Following the theory, the abstracted protocols for the Photo Sharing scenario are illustrated in Figure 13. The second step is the functional matching, i.e., check whether they have some complementary coordination policies. In this scenario, the IB producer is able to simulate the P2P consumer (under complementarity of actions), i.e., right branch of the LTS in Figure 13. The left branch, outside the dashed line, has to be discarded since it is not common with the producer application (while being common with the server of the IB application). Then, the coordination policies that IB producer and P2P consumer share are exactly the consumer's ones.

In this case, only the producer has third parties language actions and then the mediator is made by the part that translates and coordinates the common language and the part that simulates the environment by forwarding from and to it. Hence, with the application of the theory to the scenario, we obtain the connector as shown in Figure 14.

We recall (as already sketched in Section 2.2) that the high level functionalities of the various applications are the following. Taking the producer perspective (1) *authentication* - for the IB producer only -, (2) *upload of photo*, and (3) *download of comments*, while taking the consumer perspective: (i) *download of photo*, and (ii) the *upload of comments*.



**Fig. 14.** Behavioural description of the Mediating Connector for the Photo Sharing example (IB photo producer of Figure 3 a) and P2P Photo Sharing of Figure 2)

The mediator allows the interaction between the two different Photo Sharing applications by (A) manipulating and forwarding the conversations from one protocol to the other and (B) forwarding the interactions between the producer and its server. In the following, we also refer to the Basic Mediator Patterns [66] used to detect and solve the mismatches.

- The IB producer implements the authentication with the action “*Authenticate*” while the P2P does not include such functionality, i.e., there is no semantically correspondent action in the P2P application (the complementary action is in the IB server – third parties communication). Then, in this case, the mediator has to forward the interactions from the producer to its server (case (B) above).
- The IB producer implements the upload of photo with the sequence of actions “*UploadPhoto . Acknowledge*” where the former action sends both photo metadata and file and the latter models the reception of an acknowledgment. The corresponding download of photo implemented by the P2P is the sequence of actions “*PhotoMetadata . PhotoFile*”. Hence, although the actions are semantically equivalent, they do not synchronize. In order to

solve the mismatches among the upload/download of photo, the mediator has to *split* “*UploadPhoto*” into “*PhotoMetadata . PhotoFile*” and then *produce* the “*Acknowledge*”. To detect and solve the mismatches the mediator can respectively leverage on *message splitting pattern* and *message producer pattern* [66]. In this case, the mediator (case (A) above) manipulates and forwards the actions from one protocol to the other.

- The P2P implements the upload of comments with the action “*PhotoComment*” while the IB producer implements the respective download of comments with the action “*CommentPhoto*”. In order to solve the described mismatch the mediator has to perform a properly translation of “*PhotoComment*” into “*CommentPhoto*”.

In order to detect and solve the described signature mismatch, the mediator can use the *message translator pattern* [66]. In this case (case (A) above), the mediator manipulates and forwards the conversations from one protocol to the other.

Note that the building of a connector can be slightly different according to the kind of protocols to be mediated.

If the control of a protocol  $P$  is characterized by both send and receive actions, then the mediator will (i) receive an action(s) from  $P$ , (ii) properly manipulate it(them), and (iii) send it(them) to the compatible protocol  $Q$  of  $P$  and vice versa with switched roles between  $P$  and  $Q$ . Hence the mediator will synchronize with  $P$  ( $Q$  resp.) to both receive or send messages.

Instead, if the control of protocol  $P$  ( $Q$  resp.) is only characterized by send actions (i.e., it implements the client role only) then the mediator will only receive actions from  $P$  ( $Q$ ).

## 5 Related works

In this chapter we introduced application-layer connectors by referring to both coordinators and mediators. According to these two notions of connector, in this section, we discuss related work in the areas of both automatic coordinator synthesis (Section 5.1) and automatic mediator synthesis (Section 5.2). Indeed, since a mediator can be also seen as a coordinator that enables communication, these works are all related to the automatic mediator synthesis.

### 5.1 Automatic Synthesis of Coordinators

The architectural approaches to correct and automatic coordinator synthesis presented in Section 3 are related to a large number of other problems that have been considered by researchers over the past two decades. For the sake of brevity we mention below only the works closest to those approaches. The most strictly related approaches are in the “*scheduler synthesis*” research area. In the discrete event domain they appear as “*supervisory control*” or “*discrete controller synthesis*” problem [16,60] addressed by Wonham, Ramadge et al. In very general

terms, these works can be seen as an instance of the interoperability problem as (re)phrased in Section 3. However, the application domain of these approaches is sensibly different from the software component domain. Dealing with software components introduces a number of further problematic dimensions to the original synthesis problem. In the scheduler synthesis approaches the possible system executions are modeled as a set of event sequences, and the system specification describes the desired executions. The role of the supervisory controller is to interact with the system in order to meet system specification. The aim of these approach is to restrict the system behavior so that it is contained in a desired behavior, called the *specification*. To do this, the system is constrained to perform events only in strict synchronization with another system, called the *supervisor* (or *controller*). This is achieved by automatically synthesizing a suitable supervisor with respect to the system specification. In contrast to our method, there is one main assumption to deal with deadlocks: in order to automatically synthesize a *supervisor* which avoids deadlocks, they need to consider a specification of the deadlocking behaviors of the base system (i.e., the event sequences that might cause deadlocks). This is a problem because, for large systems, the designers might not know the deadlocking behaviors since they might be unpredictable.

Other works that are related to our approach appear in the *model checking of software components* context in which *compositional reachability analysis* [33] and *automatic assumption generation* [34] techniques are largely used. In [33] Giannakopoulou, Kramer and Cheung described a compositional approach to efficiently perform functional analysis of distributed systems. They validate the behavior of a distributed system with respect to specified safety and liveness properties. The hierarchical software architecture imposed on the system model to be validated allows them to reduce its size. In fact, by exploiting the system hierarchical structure, they are able to check its subsystems against the specified properties. At this point, each subsystem can be minimized in order to be modeled as a single component and the analysis is incrementally carried on. In contrast to our method they are able to minimize the model of the global system by performing efficient analysis. However, the problem faced by their approach is limited to analysis while our technique goes beyond *analyzing* functional properties of a system by also considering the problem of *automatically forcing* the system to exhibit only deadlock-free and specified behaviors. In [34] Giannakopoulou, Pasareanu and Barringer faced a problem that can be seen as an instance of the general problem (re)formulated in Section 3. In the case of these approaches, the treated problem can be formulated as follows: given a component  $C$  and a desired behavior  $B$ , find an environment  $E$  for  $C$  in such a way that  $E(C) \equiv B$  under an appropriate notion of equivalence. In this approach when model checking a component against a property, the algorithm returns one of the following three results: i) the component satisfies the property for any environment; ii) the component violates the property for any environment; or finally iii) an automatically generated set of assumptions that characterizes exactly those environments in which the component satisfies the property. The

difference with our approach is that they automatically synthesize the assumptions that represent the *weakest* environment in which the component satisfies the specified properties. That is, they deal with only two components: i) one actual component and ii) its environment. Moreover, they find an environment in such a way that the specified property is ensured but they do not guarantee the property for any possible environment.

Promising formal techniques for the compositional analysis of component-based design have been developed in [24,55]. The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioral properties. In [24], De Alfaro and Henzinger use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for components interaction. The purpose of this work is different from ours. The authors check that two components have compatible interfaces if a legal environment letting them correctly interact there exists. Each legal environment is an adaptor for the two components. They provide only a consistency check among components interfaces. That is they do not deal with automatic synthesis of component interface adaptors (i.e., automatic synthesis of legal environments). However in [55] De Alfaro, Henzinger, Passerone and Sangiovanni-Vincentelli use a game theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. In contrast to the works described in Section 3, with respect to deadlock-freedom, the specification of the converter’s requirements is assumed to be correct. Thus if, e.g., the specification would erroneously introduce deadlocks, they would not be prevented by the converter that it is synthesized in order to be completely compliant to its requirements specification. In other words, a *deadlock preventing* specification of the requirements to be satisfied by the adaptor has to be provided by delegating to the user the non-trivial task of specifying it.

Our research is also related to work in the area of protocol adaptor synthesis developed by Yellin and Strom [88]. The main idea is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components by means of adaptors. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of interaction behavior that our synthesis approach supports. Moreover, they require a formal specification of the adaptor dictating, for example, a mapping function among events of different components. Although requiring this kind of specification enhances applicability of their approach respect to the one described in Section 3, it is in contrast with our need to be as automatic as possible. In fact even if other kinds of techniques to specify the adaptor are possible, providing the adaptor specification requires to know too many implementation details thus missing part of the goals of the work presented in Section 3. However,

if we assume to have as input that detailed adaptor specification, our approach can be used to deal with the kind of incompatibilities that Yellin and Strom face in their work. In [7,71], we extended the approach described in Section 3.1 in order to not only restrict the coordinator behavior but also augmenting it in order to consider also such incompatibilities.

In other work from Bracciali, Brogi and Canal [15,20], in the area of component adaptation, it is shown how to automatically generate a concrete adaptor from: (i) a specification of component interfaces, (ii) a partial specification of the components interaction behavior, (iii) a specification of the adaptation in terms of a set of correspondences between actions of different components and (iv) a partial specification of the adaptor. The key result is the setting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior. Analogously to the work of Yellin and Strom, although this work provides a fully formal definition of the notion of component adaptor, its application domain is different from our. Since, in specifying a system, we want to maintain a high abstraction level, assuming a specification of the adaptation in terms of a set of correspondences between methods (and their parameters) of two components requires to know many implementation details (about the adaptation) that we do not want to consider in order to synthesize the adaptor.

Concerning the research underpinning the SYNTHESISRT tool, a related work in synchronous programming is the synchronizing of different clocks. In [23], each input and output port is associated with a periodic clock. Adaptation is performed at the level of each connection between ports using finite buffers. It is sufficient to look at the clocks of two connected ports and to introduce a delay by interposing a *node buffer* between the two ports. In the context of the work described in Section 3.4, adaptation must be performed at the component level by taking into account several dimensions of the specification: the component clock, the interaction protocol, the latency, duration, and controllability of each action. For this reason, introducing delays is not sufficient and, e.g., the reordering or inhibition of actions is also required.

## 5.2 Automatic Synthesis of Mediators

The automatic synthesis of application-layer mediators presented in Section 4 relates to a wide number of works in the literature within different research areas, beyond the ones discussed in Section 5.1. The theory concentrated on the interoperability problem between heterogeneous protocols within the UbiComp environment.

UbiComp was proposed by Mark Weiser in Nineties [81] [80] as the direction for development of technology in the twenty-first century. But the early basics for this new philosophy were created in 1988 as “tabs, pads and boards” [82]. One of the key principles of UbiComp is to make the computer able to vanish in the background to increase their use making it in an efficient and invisible manner to users. UbiComp suggests the ability for users to enter the

environment in a natural way, without being a priori aware of who or what populates it. Furthermore, the user should be able to use the services available using its devices without complex procedures and manual configurations. The currently available technologies and computations have not yet reached the maturity required by the ubiquitous paradigm due to the fact that they are still tied and dependent on the underlying layers although we can qualify them as ubiquitous. The ubiquitous vision fits perfectly with our idea of mediator. Each entity, indeed, maintains its own characteristics (and diversities), being able to communicate and cooperate with the others without having any prior knowledge of them thanks to the support provided by the mediators that masks divergencies making them appear homogeneous.

*Interoperability* and *mediation* have been investigated in several contexts, among which integration of heterogeneous data sources [85,84], software architecture [32], architectural patterns [18], design patterns [31], patterns of connectors [83,68], Web services [11,22,70,45,38], and algebra to solve mismatches [26] to mention a few.

In particular the interoperability/mediation of protocols have received attention since the early days of networking. Indeed many efforts have been done in several directions including for example formal approaches to *protocol conversion* [19,44,53], and their extension towards reducing the algorithmic complexity of protocol conversion [43].

A work strictly related to the mediators presented in this chapter is, again, the work by Yellin and Strom [88] discussed in Section 5.1. With respect to our mediator synthesis approach, this work prevents to deal with ordering mismatches and different granularity of the languages (one send-many receive and many send-one receive mismatches [66]).

Recently, with the emergence of *Web services* and advocated universal interoperability, the research community has been studying solutions to the *automatic mediation of business processes* [78,77,50,86]. They differ with respect to: (a) a priori exposure of the process models associated with the protocols that are executed by networked resources, (b) knowledge assumed about the protocols run by the interacting parties, (c) matching relationship that is enforced. However, most solutions are discussed informally, making it difficult to assess their respective advantages and drawbacks.

This highlights the needed for a new and formal foundation for mediating connectors from which protocol matching and associated mediation may be rigorously defined and assessed. These relationships should be automatically reasoned upon, thus paving the way for on the fly synthesis of mediating connectors. To the best of our knowledge, such an effort has not been addressed in the Web services and Semantic Web area although proposed algorithms for automated mediation manipulates formally grounded process models.

Within the Web Services research community, a lot of work has been also devoted to *behavioral adaptation* which has been actively studying this problem.

Among these works, and related to our, there is [51]. It proposes a *matching approach* based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioral descriptions. Our matching is driven by the ontology as described in Section 4 and in [67,36].

Moreover, recently the Web services community has been also investigating how to actually support *service substitution* so as to enable interoperability with different implementations (e.g., due to evolution or provision by different vendors) of a service. While early work has focused on semi-automated, design-time approaches [50,59], latest work concentrates on automated, run-time solutions [25,21]. The work [25] addresses the interoperability problem between services and provide experimentation on real Web2.0 social applications. They propose a technique to dynamically detect and fix interoperability problems based on a catalogue of inconsistencies and their respective adapters. This is similar to our proposal to use *ontology mapping* to discover mismatches and mediator to solve them. Our work differs with respect to theirs because we aim at automatically synthesizing the mediator. Instead, their approach is not fully automatic since although they discover and select mismatches dynamically, the identification of mismatches and of the opportune adapters is made by the engineer.

Our work also closely relates to [21], sharing the exploitation of ontology to reason about interface mapping and the synthesis of mediators according to such mapping. Despite these similarities, our work goes one step further by not being tight to the specific Web service domain.

Our work also closely relates to significant effort from the *semantic Web service* domain and in particular the WSMO (Web Service Modeling Ontology) initiative that defines mediation as a first class entity for Web service modeling towards supporting service composition. The resulting Web service mediation architecture highlights the various mediations levels that are required for systems to interoperate in a highly open network [70]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [22,78].

A lot of work has also been devoted to *connectors* and include a *classification framework* [48], *studies on connectors* [87,41], and *formally grounded* works on connectors. For example, [69] presents an approach for formally specifying connector wrappers as protocol transformations, modularizing them, and reasoning about their properties, with the aim to resolve component mismatches. Another formal work is [28] the authors propose mathematical techniques as foundations to develop architectural design environments that are ADL-independent. Authors of [46] present a formal specification mechanism, by a categorical semantics, for higher order connectors concept that is connectors that take a connector as parameter and deliver another as result. In [10] the authors present a formalization of software connectors. In [17] the authors present an algebra for five basic stateless connectors that are symmetry, synchronization, mutual exclusion, hiding and inaction. They also give the operational, observational and denotational semantics and a complete normal-form axiomatization. The presented connectors can be composed in series and in parallel. A PhD thesis [9]

proposes a new connector model, in the distributed component-based context of the SOFA/DCUP project component model. The proposed model allows the description of interactions between components with a semi-automatic generation of the corresponding code.

## 6 Conclusion and Future Perspectives

Automated and on-the-fly interoperability is a key requirement for heterogeneous protocols within ubiquitous computing environments where networked systems *meet dynamically* and need to *interoperate without a priori knowledge* of each other. Although numerous efforts have been done in many different research areas, such kind of interoperability is still an open challenge.

In CONNECT, we concentrated on the automatic synthesis of mediators between compatible protocols which enables them to communicate.

We proposed *rigorous techniques* to automatically reason about and compose the behavior of networked systems that aim at fulfilling some goal by connecting to other systems.

The reasoning serves to find a way to achieve communication -if it is possible- and to build the related mediation solution. Our current work put the emphasis on “the elicitation of a way to achieve communication” while it can gain from more practical treatment of similar problems in the literature like the coordinators synthesis or, e.g., converters or adaptors. In particular, we contributed with:

- the design of a comprehensive mediator synthesis process described in [64];
- a set of mediator patterns which represent the building blocks to tackle in a systematic way the protocol mediation. This led us to devise a complete characterization of the protocol mismatches that we are able to solve by our connector synthesis process and to define significant mediator patterns as solution to the classified problems. This is reported in [66] and is revised and extended in [65];
- a formalization of a theory of emerging mediating connectors which includes related automated model-based techniques and tools to support the devised synthesis process. The theory rigorously characterizes: (i) application layer protocols, (ii) their abstraction, (iii) the conditions under which two protocols are functionally matching, (iv) the notion of interoperability between protocols based on the definition of the functional matching relationship, and (v) the mapping, i.e., the synthesis of the mediator behavior needed to achieve protocol interoperability under functional matching. This is illustrated in Section 4 as well as in [67,36,64].
- A combined approach including the theory and a monitoring system towards taking into account also non-functional properties reported in [14].

In the following we discuss *future work perspectives*. The theory of mediators proposed in this chapter (1) clearly defines the interoperability problem, (2) shows the feasibility of the automated reasoning about protocols, i.e., functional

matching, and (3) shows the feasibility of the automated synthesis of abstract mediators under certain assumptions. In the future we plan to:

- implement the theory algorithms in order to automatize the mediator generation. In this direction, we are currently working on on-the-fly reasoning about interoperability using ontology-based model checking [12];
- extend the theory of mediators so to have a comprehensive framework for dealing also with middleware layer protocols and data, in addition to application layer protocols. This is currently being investigated [13];
- study run-time techniques towards efficient synthesis;
- scale the synthesis process. The current theory is described considering only two protocols but extending it to an arbitrary number  $n$  of protocols seems not to be problematic. The protocol abstraction step developed within the devised process represents a first attempt in this direction by reducing the size of the behavioral models of the NSs to be connected;
- extend the validation of the theory on other real world applications. It would possibly help in tuning the theory, if needed, and in refining the borders among which the theory works;
- translate the synthesized connector model into an executable artefact that can be deployed and run in the network for actual enactment of the connectors, as studied in the CONNECT project. This also requires devising the runtime architecture of CONNECTors (see Deliverables D1.1 [1] and D1.2 [2] of CONNECT) by investigating the issue of generation of code versus interpretation of the connector model. First results in this direction are in Deliverable D3.2 [3];
- ensure dependability. While preliminary results towards this aim have been described in [14], we aim to take into account both functional interoperability and non-functional interoperability *during* the process. Indeed we would include also the modeling of non-functional aspects, together with their respective matching and mapping reasoning.
- relax some assumptions, towards a dynamic environment, and manage the consequent uncertainty. For example, we aim at integrating with complementary works ongoing within the CONNECT project so as to develop an overall framework enabling the dynamic synthesis of emergent connectors among networked systems. Instances of complementary works are (i) learning techniques to dynamically discover the protocols (instead of assuming them given) that are run in the environment; (ii) data-level interoperability (instead of assuming the ontology given) to elicit the data mappings. This may rise the problem of dealing with partial or erroneous specifications.

## References

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

2. CONNECT consortium. CONNECT Deliverable D1.2: Intermediate Connect Architecture. FET IP CONNECT EU project, FP7 grant agreement number 231167, <http://connect-forever.eu/>.
3. CONNECT consortium. CONNECT Deliverable D3.2: Reasoning about and Harmonizing the Interaction Behavior of Networked Systems at Application- and Middleware- Layer. FET IP CONNECT EU project, FP7 grant agreement number 231167, <http://connect-forever.eu/>.
4. ITU Telecommunication Standardisation sector, ITU-T recommendation Z.120. Message Sequence Charts. (MSC'96). Geneva.
5. A. Arnold. Finite Transition Systems. In *International Series in Computer Science*. Prentice Hall International (UK), 1989.
6. M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, pages 784–787, 2007. IEEE Computer Society. DOI REF: <http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.84>.
7. M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of “correct” adaptors for protocol enhancement in component based systems. In *Proceedings of the 1st International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'04) at FSE' 04*, pages 79–86, 2004.
8. M. Autili, L. Mostarda, A. Navarra, and M. Tivoli. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software*, 81(12):2210–2236, 2008.
9. D. Balek. *Connectors in Software Architectures*. PhD thesis, Charles University, May 2002.
10. M. A. Barbosa and L. S. Barbosa. Specifying software connectors. In *ICTAC*, pages 52–67, 2004.
11. B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adaptors for web services integration. In *proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, Porto, Portugal, pages 415–429. Springer Verlag, 2005.
12. A. Bennaceur, V. Issarny, and R. Spalazzese. On-the-fly reasoning about interoperability using ontology-based model checking. technical report, inria rocquencourt, paris., Jan. 2011.
13. A. Bennaceur, R. Spalazzese, P. Inverardi, V. Issarny, N. Georgantas, and R. Saadi. Model-based mediators for dynamic-adaptive connectors. Technical report, INRIA Paris-Rocquencourt, France, 2011.
14. A. Bertolino, P. Inverardi, V. Issarny, A. Sabetta, and R. Spalazzese. On-the-fly interoperability through automated mediator synthesis and monitoring. In *ISoLA 2010, Part II, LNCS 6416*,., pages 251–262. Springer, Heidelberg, 2010.
15. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74, January 2005.
16. B. Brandin and W. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2), 1994.
17. R. Bruni, I. Lanese, and U. Montanari. A basic algebra of stateless connectors. *Theor. Comput. Sci.*, 366(1):98–120, 2006.
18. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.
19. K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communications*, 8(1):127–142, 1990.

20. C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Software Eng.*, 34(4):546–563, 2008.
21. L. Cavallaro, E. D. Nitto, and M. Pradella. An automatic approach to enable replacement of conversational services. In *ICSOC/ServiceWave*, 2009.
22. 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.
23. A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. Synchronization of periodic clocks. In *Proc. of the 5th EMSOFT*, 2005.
24. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, 2001.
25. G. Denaro, M. Pezzé, and D. Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of ESEC/FSE 2009*. ACM Press, 2009.
26. M. Dumas, M. Spork, and K. Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Business Process Management*, pages 65–80, 2006.
27. P. Feiler, R. P. Gabriel, J. Goodenough, R. Lingerand, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. 2006.
28. J. L. Fiadeiro, A. Lopes, and M. Wermelinger. Theory and practice of software architectures. Tutorial at the 16th IEEE Conference on Automated Software Engineering, San Diego, CA, USA, Nov. 26-29, 2001., 2001.
29. A. Finkel. The minimal coverability graph for Petri nets. In *Proc. of the 12th APN*, volume 674 of *LNCS*, 1993.
30. G. Blair et al. Introduction to Interoperability. In *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*. *LNCS series*. 2011.
31. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Resuable Object-Oriented Software*. Addison-Wesley Professional, 1995.
32. D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.
33. D. Giannakopoulou, J. Kramer, and S. C. Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6:7–35, 1999.
34. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engg.*, 12:297–320, 2005.
35. B. Intrigila, P. Inverardi, and M. V. Zilli. A comprehensive setting for matching and unification over iterative terms. *Fundam. Inform.*, 39(3):273–304, 1999.
36. P. Inverardi, V. Issarny, and R. Spalazzese. A theory of mediators for eternal connectors. In *Proceedings of ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Part II.*, volume 6416, pages 236–250. Springer, Heidelberg, 2010.
37. P. Inverardi and M. Nesi. Deciding observational congruence of finite-state ccs expressions by rewriting. *Theor. Comput. Sci.*, 139(1-2):315–354, 1995.

38. F. Jiang, Y. Fan, and X. Zhang. Rule-based automatic generation of mediator patterns for service composition mismatches. In *Proceedings of the 2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, pages 3–8, Washington, DC, USA, 2008. IEEE Computer Society.
39. Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *Knowl. Eng. Rev.*, 18(1):1–31, January 2003.
40. 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.
41. S. Kell. Rethinking software connectors. In *SYANCO '07: International workshop on Synthesis and analysis of component connectors*, pages 1–12, New York, NY, USA, 2007. ACM.
42. R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.
43. R. Kumar, S. Nelvagal, and S. I. Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems*, 7(3), 1997.
44. S. S. Lam. Correction to "protocol conversion". *IEEE Trans. Software Eng.*, 14(9):1376, 1988.
45. 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.
46. A. Lopes, M. Wermelinger, and J. L. Fiadeiro. Higher-order architectural connectors. *ACM Trans. Softw. Eng. Methodol.*, 12(1):64–104, 2003.
47. J. Magee and J. Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.
48. N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM Press.
49. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
50. 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.
51. H. R. Motahari Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th international conference on World wide web, WWW '10*, pages 731–740, New York, NY, USA, 2010. ACM.
52. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.
53. K. Okumura. A formal protocol conversion method. In *SIGCOMM*, pages 30–37, 1986.
54. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering, ICSE '98*, pages 177–186, 1998.
55. R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ICCAD '02*, pages 132–139, 2002.

56. P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for designing and verifying architectural specifications. *IEEE Trans. Softw. Eng.*, 35:325–346, May 2009.
57. P. Pelliccione, M. Tivoli, A. Bucchiarone, and A. Polini. An architectural approach to the correct and automatic assembly of evolving component-based systems. *Journal of Systems and Software*, 81(12):2237–2251, 2008.
58. G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
59. S. Ponnekanti and A. Fox. Interoperability among independently evolving Web services. In *Proc. ACM/IFIP/USENIX Middleware Conference*, pages 331–351, 2004.
60. P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *Siam J. Control and Optimization*, 25(1), 1987.
61. P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 1(77), 1989.
62. J. Reynolds. Transformational systems and the algebraic structure of atomic formulas machine intelligence, edinburgh university press, usa, vol. 5, pp. 135–151, 1970.
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. R. Spalazzese. *A Theory of Mediating Connectors to achieve Interoperability*. PhD thesis, University of L’Aquila, April 2011.
65. R. Spalazzese and P. Inverardi. Components interoperability through mediating connector pattern. In *WCSI 2010, arXiv:1010.2337; EPTCS 37, 2010*, pp. 27–41.
66. R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *ECSCA*, pages 335–343, 2010.
67. 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/ECSCA 2009)*, pages 345–348, 2009.
68. B. Spitznagel. *Compositional Transformation of Software Connectors*. PhD thesis, Carnegie Mellon University, May 2004.
69. B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *ICSE*, pages 374–384, 2003.
70. 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.
71. M. Tivoli and M. Autili. Synthesis, a tool for synthesizing correct and protocol-enhanced adaptors. *RSTI - L’objet, Coordination and Adaptation Techniques*, 12(1):77–103, 2006.
72. M. Tivoli, P. Fradet, A. Girault, and G. Goessler. Adaptor synthesis for real-time components. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), member of ETAPS’07*, volume 4424 of *LNCS*, pages 185–200. Springer-Verlang Berlin/Heidelberg, 2007.
73. M. Tivoli and P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming*, 71(3):181–212, 2008.
74. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proceeding of the 23rd IEEE International Conference on Software Engineering (ICSE’01)*, 2001.

75. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, 2001.
76. V. Issarny et al. Middleware-layer Connector Synthesis. In *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*. LNCS series. 2011.
77. 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.
78. R. Vaculín and K. Sycara. Towards automatic mediation of OWL-S process models. *Web Services, IEEE International Conference on*, 0:1032–1039, 2007.
79. S. M. Watt. Algebraic generalization. *SIGSAM Bull.*, 39(3):93–94, 2005.
80. M. Weiser. The computer for the 21<sup>st</sup> century. *Scientific American*, Sep. 1991.
81. M. Weiser. Hot Topics: Ubiquitous Computing. *IEEE Computer*, oct 1993.
82. M. Weiser. Ubiquitous computing. <http://sandbox.xerox.com/ubicomp/>, 1996.
83. M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Trans. Softw. Eng.*, 24(5):331–341, 1998.
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. Woollard and N. Medvidovic. High performance software architectures: A connector-oriented approach. In *Proceedings of the Institute for Software Research Graduate Research Symposium, Irvine, California*, June 2006.
88. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19, 1997.