Journal of Computer and System Sciences 104 (2019) 17–40

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Automated synthesis of application-layer connectors from automata-based specifications

Marco Autili [c], Paola Inverardi [c], Romina Spalazzese [a,b], Massimo Tivoli [c,*], Filippo Mignosi [c,d]

[a] Department of Computer Science and Media Technology, Malmö University, Nordenskiöldsgatan 1, SE-211 18 Malmö, Sweden
[b] Internet of Things and People Research Center, Malmö University, Sweden
[c] Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila, Via Vetoio, 67100 L'Aquila, Italy
[d] Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Via Archirafi 34, 90123 Palermo, Italy

ARTICLE INFO

Article history:
Received 17 July 2015
Received in revised form 7 March 2019
Accepted 13 March 2019
Available online 28 March 2019

Keywords:
Automated mediator synthesis
Interoperability
Protocols
Heterogeneous applications
Communication & coordination
Protocol mismatches

ABSTRACT

Ubiquitous and Pervasive Computing, and the Internet of Things, promote dynamic interaction among heterogeneous systems. To achieve this vision, interoperability among heterogeneous systems represents a key enabler, and mediators are often built to solve protocol mismatches. Many approaches propose the synthesis of mediators. Unfortunately, a rigorous characterization of the concept of interoperability is still lacking, hence making hard to assess their applicability and soundness. In this paper, we provide a framework for the synthesis of mediators that allows us to: (i) characterize the conditions for the mediator existence and correctness; and (ii) establish the applicability boundaries of the synthesis method.

1. Introduction

Ubiquitous and Pervasive Computing, as well as the Internet of Things, promote the creation of a computing environment where heterogeneous software systems seamlessly interact to achieve common tasks [1–3]. When composing heterogeneous systems, one of the main problems is to achieve their interoperability by solving protocol mismatches. The term protocol refers to *behavioral* protocols or *interaction* protocol or *observable* protocols. A protocol is the sequences of actions visible at the interface level which a system performs while interacting with other systems. In this paper we consider *application-layer* protocols. They model the sequences of actions that need to be performed to achieve a business-level task, e.g., accessing a bank account service after performing login and getting authorized.

Addressed issue. The problem we address in this paper is to *automatically achieve the interoperability among an arbitrary number n of heterogeneous protocols, where n is equal to or greater than two.*

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 "complementary" send/receive actions. The *coordination* is expressed by the achievement of a goal, i.e., two systems succeed in coordinating if they are able to achieve the common goal(s) through suitable synchronization of their actions.

* Corresponding author.
E-mail addresses: marco.autili@univaq.it (M. Autili), paola.inverardi@univaq.it (P. Inverardi), romina.spalazzese@mah.se (R. Spalazzese), massimo.tivoli@univaq.it (M. Tivoli), filippo.mignosi@univaq.it (F. Mignosi).

https://doi.org/10.1016/j.jcss.2019.03.001
0022-0000/

Communications requiring complex protocol interaction can be regarded as a form of coordination, i.e., a composition of basic synchronizations. In fact, application level protocols introduce a notion of communication that can go far beyond a single basic synchronization thus requiring well defined sequences of synchronizations to be achieved.

In order to enable communication and correct coordination between heterogeneous protocols, we propose automated synthesis of application-layer *mediators*. A mediator is a specific connector that includes at the same time a communication and a coordination solution to solve protocols mismatches.

State of the art and motivations. Interoperability problems, and specific notions of connector that can be adopted to (partly) solve them, have been the focus of extensive studies of different research communities. Protocol interoperability comes from the early days of networking and different efforts, both theoretical and practical, have been done to address it in several areas including: protocol conversion [4–7], component adaptors [8,9], Web services mediation [10–16], theories of connectors [17,18], wrappers [19], bridges, and interoperability platforms, to mention a few.

Despite the existence of numerous solutions in the literature, most of them are focused on coordinator synthesis and less effort has been devoted to the automatic synthesis of mediators. In particular, most of the approaches: (i) assume the communication problem solved by considering protocols already able to communicate; (ii) are informal, hence making automatic reasoning impossible; (iii) follow a semi-automatic process for the mediator synthesis requiring human intervention; (iv) consider only few possible mismatches.

Contribution and aim of the work. To overcome the above mentioned shortcomings of existing solutions, we define a model-based framework for mediators together with an automated approach for the mediator synthesis. The approach includes three steps: *abstraction*, *matching*, and *synthesis*. Our mediator synthesis method can be applied to an arbitrary number of $n \geq 2$ protocols.

This paper aims to provide a solid foundation for the formal characterization of the mediator synthesis problem. In particular, we provide a rigorous characterization of the interoperability notion and, hence, of the synthesis problem. This characterization relies on the use of both *labeled transition systems* to model protocols and *ontologies*. It allows us to: (i) define the conditions that ensure mediator existence; (ii) show that our synthesis method is sound and the synthesized mediator is correct by construction; and (iii) establish the applicability boundaries of the method.

The work presented in this paper substantially extends our work [20] where we just provided an overview of the approach.

Roadmap. The paper is organized as follows. Section 2 sets the context of the work reported in this paper and introduces the *purchase order scenario* that we use as running example throughout the evolution of the paper. Section 3 provides an overview of the synthesis method. Section 4 formalizes the method, illustrates it at work on the use case scenario, and proves its correctness. Section 5 discusses related works by also providing a detailed comparison with the works closest to ours. Section 6 discusses (i) final remarks, (ii) limitations of the work by also mentioning possible future extensions that will allow to cope with them, and (iii) further future directions.

2. Setting the context

The main assumption of our connector synthesis technique is related to the possibility of characterizing the interaction protocol of a system by means of an automata-based specification such as *Labeled Transition Systems* (LTSs) [21]. This assumption is supported by the increasing proliferation of techniques for software model elicitation that generate automata (see [22–27], just to cite a few).

We model system interaction by means of behavioral protocols that specify the order in which input and output actions are performed while the systems interact with their environment, i.e., other systems. Systems communicate through messages exchange. That is, from the point of view of one system, input actions model the receiving of a message from another system; output actions model the sending of a message to another system. For the purposes of this paper, we model synchronous communication only. Note that this is not a limitation because it is well-known that with the introduction of a buffer component we can also model an asynchronous system by a synchronous one [28,29].

As already introduced, we focus on the automated synthesis of application-layer connectors, also referred to as *mediators*, which mediate the interactions among heterogeneous systems to achieve their interoperability, i.e., *communication* and *correct coordination*.

In the following, we describe an example showing interoperability problems that can be solved by using our mediators.

2.1. The need for mediators: the purchase order scenario

This section introduces a use case concerning the so called *Purchase Order Mediation scenario* from the Semantic Web Service (SWS) Challenge [30,31]. The scenario considers two *prosumer*[1] Web Services (WSs) using different protocols: the *Moon Service* (MS) and the *Blue Client* (BC). When dealing with mediation issues, the scenario is representative of a number of service-based applications.

[1] A prosumer is both a *consumer* and a *provider* of service operations.

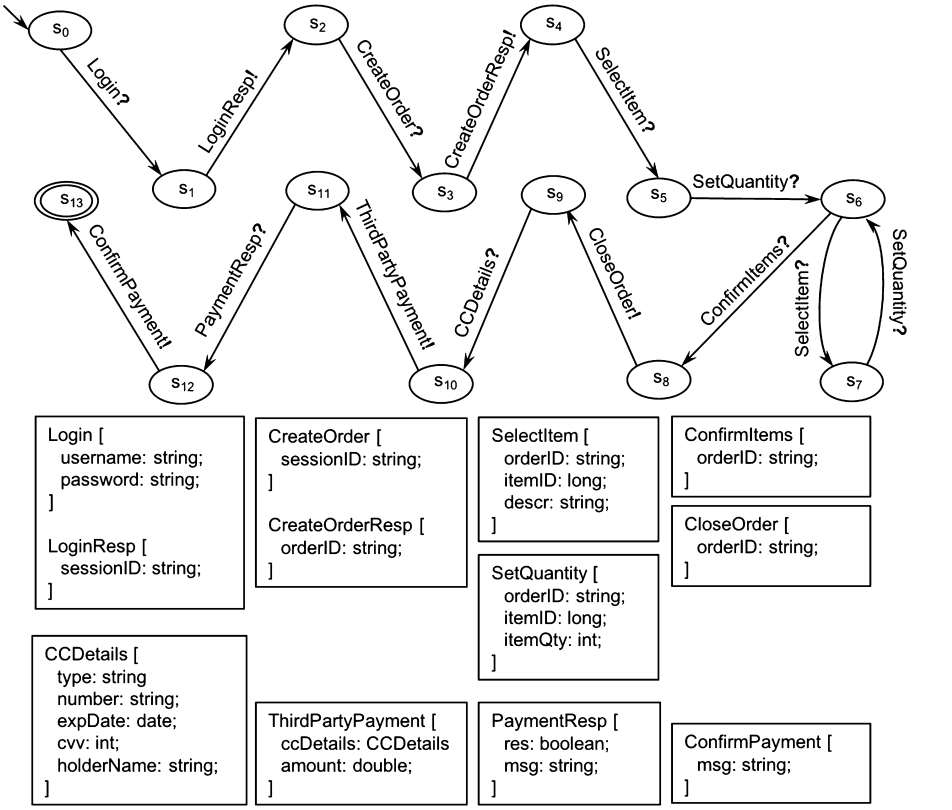

Fig. 1. Behavioral protocol of MS.

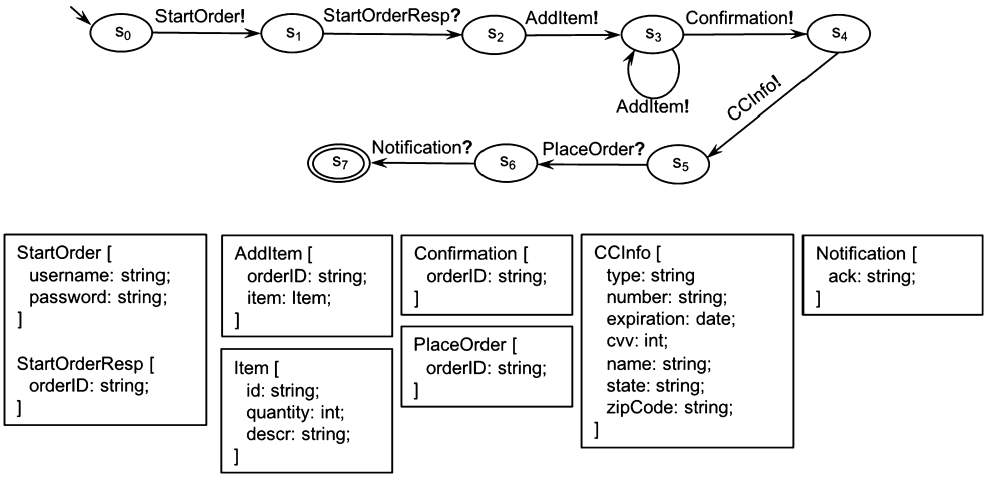

Fig. 2. Behavioral protocol of BC.

Figs. 1 and 2 show the LTS-based specification of the MS protocol and the BC protocol, respectively. The adopted LTS-based specification is formalized in Section 4. Informally, a transition from a state s to a state s' labeled with $m!$ models the sending of a message m, a transition labeled with $m?$ models the receiving of m. The states labeled with s_0 are *initial* states, double circled states are *final*, and any sequence of actions starting from the initial state and leading to a final state is a *trace*. Typically, final states are used to model the achievement of some application-level business goal and, hence, traces represent meaningful conversations with the environment. The content of the boxes at the bottom of both Figs. 1 and 2 shows all the information (i.e., the name plus the internal structure of the message type) concerning the message names used to label the protocol transitions. Note that on the transitions we simply used message names for the readability of the figures.

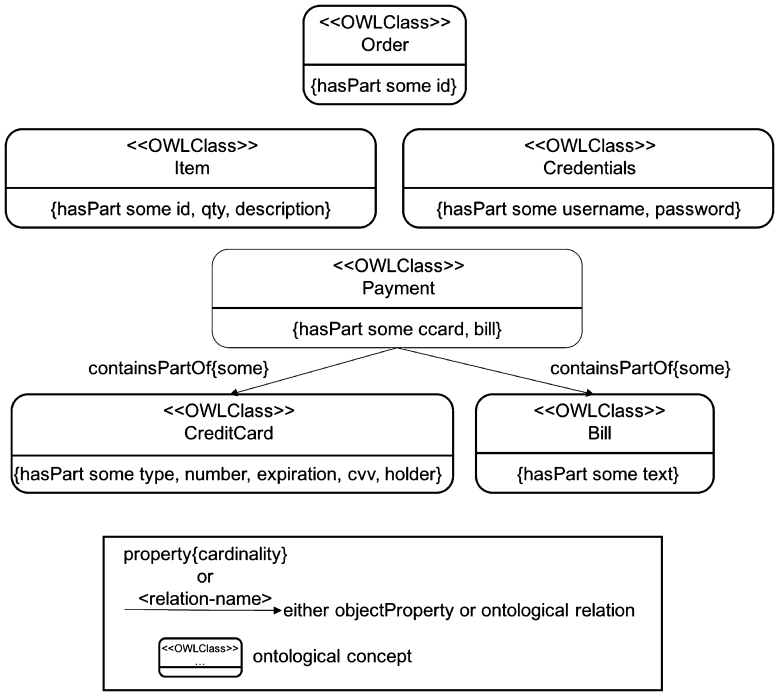

Fig. 3. Domain ontology O.

MS receives product orders according to the following protocol: after receiving the authentication request (the `Login?` message containing the customer identifier), from authorized customers, MS sends back a session unique identifier (contained in the `LoginResp!` message). MS is now ready to receive a request for creating a purchase order (`CreateOrder?` containing the session identifier). When the order is created, its identifier is sent back to the client (`CreateOrderResp!` containing the order identifier). After at least one item is selected and its quantity set (`SelectItem?` and `SetQuantity?` messages), the confirmation message, for the addition of the chosen number of selected items to the cart, can be received (`ConfirmItems?`). Then the order is closed and a notification of it is sent to the client (`CloseOrder!`). Finally, credit card details can be received (`CCDetails?`) in order to perform the order payment by means of a third-party payment system (`ThirdPartyPayment!` followed by `PaymentResp?`), and the payment confirmation is sent to the client (`ConfirmPayment!`).

BC is a client to issue purchase orders. It initiates an order by creating it upon providing authentication credentials (`StartOrder!`). After receiving the identifier of the created order (`StartOrderResp?`), BC adds one or more items to the order (`AddItem!`); each item specifies the needed quantity. Afterwards, it sends a confirmation to its server for the added items (`Confirmation!`). Then, BC sends the credit card details to perform the order payment (`CCInfo!`). After this, BC receives the notification confirming the placement of the order (`PlaceOrder?`) and, finally, it receives the payment notification (`Notification?`).

MS and BC cannot interoperate due to a number of communication and coordination mismatches [32]. Some of them are briefly discussed below.

Communication mismatches concern the semantics and granularity of the protocol actions. For instance, BC uses a single message to add an item to the order and set its quantity (`AddItem`), whereas MS expects to use two different messages, one for the item addition (`SelectItem`) and one for the quantity specification (`SetQuantity`). To solve these kinds of mismatches we assume to have some semantic knowledge modeled into an ontology and, by reasoning on it, our method infers information useful for mediation purposes. The inferred information leads to produce a mediator that performs the proper message translation.

Coordination mismatches concern the control structure of the protocols and can be solved by means of a mediator that can mediate the conversation between the two protocols so that they can actually interact. For instance, MS requires its clients to receive the confirmation for the closed order and then provide the credit card details (`CloseOrder!` followed by `CCDetails?`); whereas BC expects to get the confirmation of the closed order only after the credit card details have been sent to the server (`CCInfo!` followed by `PlaceOrder?`). Again, to solve these kinds of mismatches we reason on ontologies to synthesize a mediator that, e.g., reorders sequences of messages, splits a single message into a sequence of messages, and merges a sequence of messages into a single message.

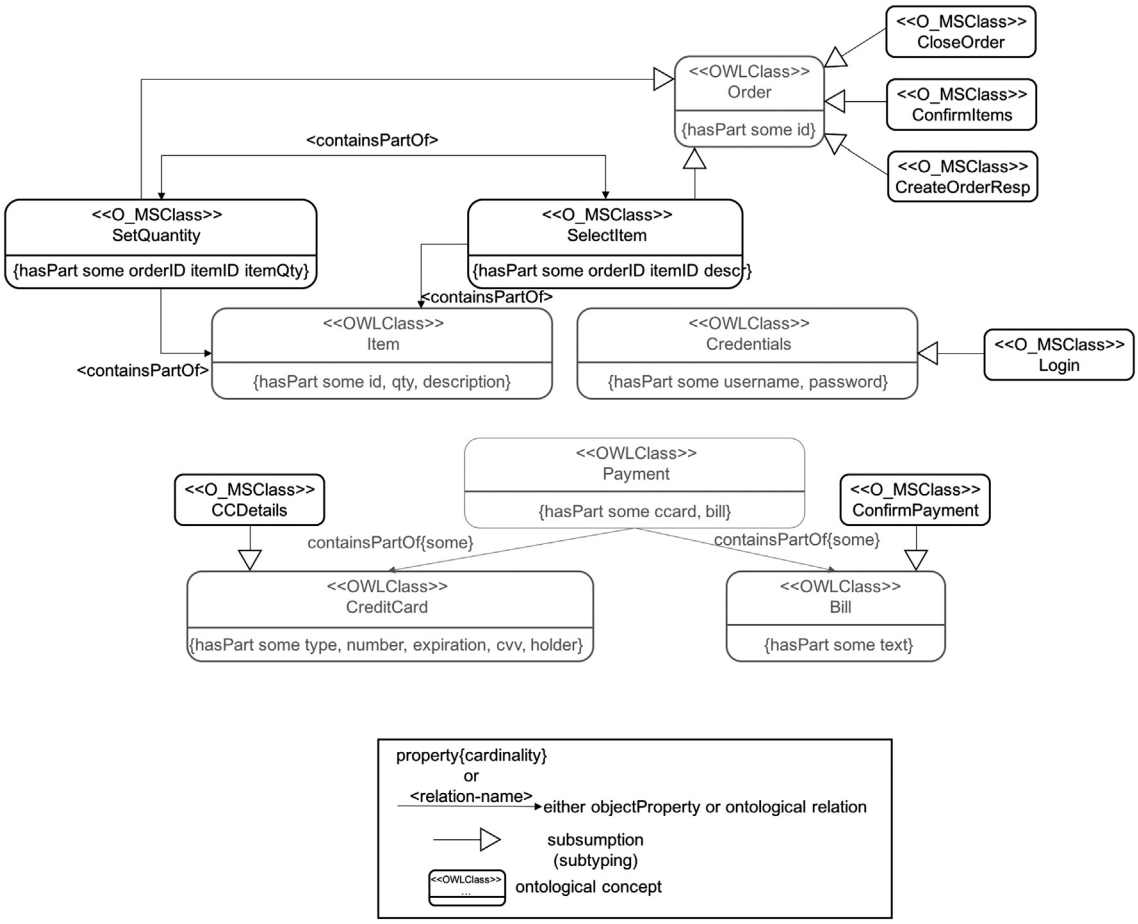

Fig. 4. Protocol ontology O_{MS}.

2.2. The role of ontologies

MS and BC are provided by the SWS-Challenge organizers and cannot be altered. However, nowadays there exist several *domain ontologies* [33] (e.g., for e-commerce domains[2]) that can serve as common descriptions of specific domains and, as such, can be shared among and referred by different applications for the purpose of producing application-specific *protocol ontologies*. Thus, by exploiting an ontology in the domain of *purchase processes*, the descriptions of MS and BC can be semantically enriched by two protocol ontologies, one for MS and one for BC.

Ontologies are expressed by using dedicated languages (e.g., OWL, DAML, OIL, RDF Schema, just to mention a few) that support specializations and extensions, while enabling automated reasoning for determining ontology mapping [34]. Ontologies account for fundamental relations between concepts [35] and, in this paper, we make use of *subsumption* and *inclusion*:

- a concept a is subsumed by a concept b if, in every model of the ontology, the set denoted by a is a subset of the set denoted by b. In other words, subsumption can be seen as a subtyping relation between concept types (i.e., b is a subtype of a).
- a concept a is included in a concept b if the latter contains the former either entirely or partially (i.e., b contains either a or part of a).

The domain ontology that we consider for our use case is denoted by O, and is shown in Fig. 3. The protocol ontologies for MS and BC are denoted by O_{MS} and O_{BC}, and are shown in Figs. 4 and 5, respectively. Note that while defining the protocol ontologies O_{MS} and O_{BC}, the protocol developer relates protocol-specific concepts of O_{MS} and O_{BC} to the ones of the domain ontology O. Informally, these (protocol-specific) concepts correspond to protocol messages and the relations

[2] http://www.heppnetz.de/projects/goodrelations/.

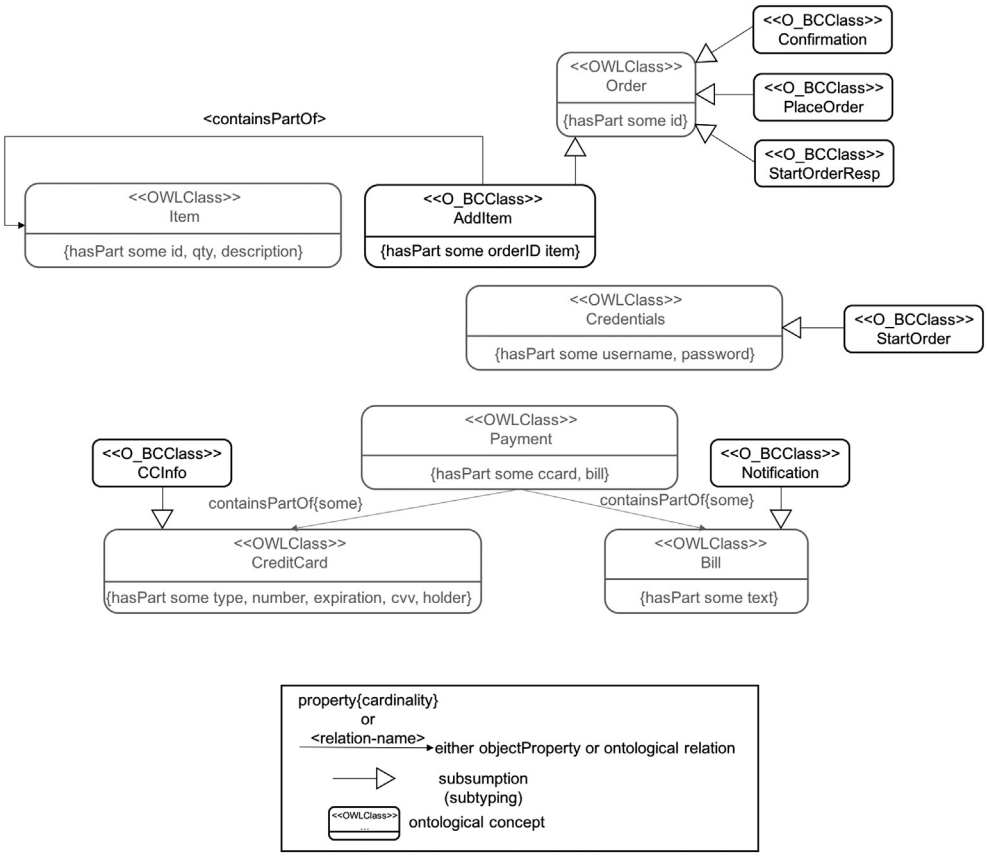

Fig. 5. Protocol ontology O_{BC}.

among them are defined according to the elements that constitute their internal structure. In Figs. 4 and 5, the portions of O referred by O_{MS} and O_{BC} are depicted in light grey in order to highlight the performed specializations and extensions.

The domain ontology in Fig. 3 intuitively specifies that, in the considered domain, there are certain concepts that are usually accounted for: purchase `Items` identified by an identifier, having a description and a quantity specified; `Orders` identified by an identifier, and `Credentials` consisting of username and password; finally, `Payments` made of `CreditCard` details and related `Bill`.

For example, by referring to the protocol ontology shown in Fig. 4, an instance of `Login` subsumes an instance of `Credentials`, i.e., the type of `Login` is a subtype of the type of `Credentials`. Furthermore, the type of `SetQuantity` includes (possibly partially) the type of `SelectItem` and vice versa. It is worth noticing also that, the type of `Item` is included in the type of both `SetQuantity` and `SelectItem`; and, similarly, the type of both `SetQuantity` and `SelectItem` is a subtype of `Order`. This last example gives an important information: given an instance of `SetQuantity` and one of `SelectItem` that are mutually included one into the other, they share the same instances of `Order` and `Item`. In Section 4.5, we will show what kind of reasoning this kind of information enables for the purposes of automated mediator synthesis.

3. Method overview

Fig. 6 shows an overview of our synthesis method as applied to an arbitrary number $n \geq 2$ of protocols.

The method takes as input the behavioral protocols of the considered systems, $P_1, \ldots, P_n$, semantically enriched by their protocol ontologies, $O_1, \ldots, O_n$, respectively. $O_1, \ldots, O_n$ refer to the domain ontology O and, hence, they share common concepts belonging to O. If $P_1, \ldots, P_n$ are not interoperable then a mediator is required, if any exists. It is automatically synthesized by performing the following three steps: *Abstraction*, *Matching*, and *Synthesis*.

Step 1 – The abstraction step takes as input a behavioral protocol P_i and its protocol ontology O_i and produces an abstraction of P_i. It is modeled by an abstract behavioral protocol P_i^A. This new protocol is abstract since its alphabet of actions refers only to the concepts in O that have been used to define O_i. These concepts are an abstraction of the messages exchanged by P_i. As a consequence, differently from $P_1, \ldots, P_n$, their abstract protocols share a common alphabet of actions. For a set of protocols to be mediated, sharing a common alphabet is a necessary condition, yet not sufficient, for checking the existence of a mediator and synthesizing it.

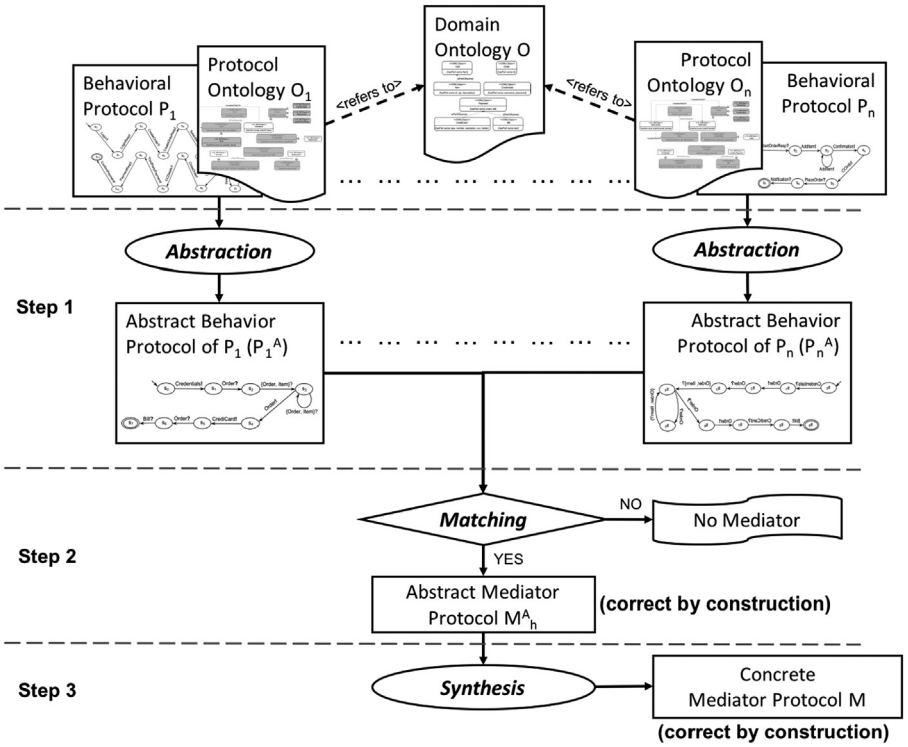

Fig. 6. Method overview.

Step 2 – By reasoning on $P_1^A, \ldots, P_n^A$, the matching step produces an abstract behavioral protocol M_h^A. It models the protocol of an abstract mediator that could serve as a basis for synthesizing the final concrete mediator. That is, if the set of traces of M_h^A is empty then there is no way to mediate the interaction of $P_1, \ldots, P_n$ to make them interoperable and, hence, a concrete mediator for $P_1, \ldots, P_n$ does not exist. Otherwise, a concrete mediator for $P_1, \ldots, P_n$ exists and it is produced by the next step.

An important consideration here concerns the ability of the mediator to perform reordering of messages and deal with cycles. As detailed in Section 4, the mediator uses a message buffer to reorder sequences of messages. By admitting protocols with cycles, unbounded (i.e., infinite state) behavior might be generated while synthesizing the abstract mediator, hence leading to an unrealizable mediator. To overcome this problem, the method allows the user to set a bound $h > 0$ on the size of the mediator buffer. Thus, rephrasing the above statement, if the set of traces of M_h^A is empty, then there is no way to correctly mediate the considered protocols by exploiting a message buffer whose size is set to h.

This means that the method is correct, although not complete, if protocols with cyclic behavior must be dealt with. Instead, it is both correct and complete for protocols without cycles.

Step 3 – Out of M_h^A, $P_1, \ldots, P_n$, and $O_1, \ldots, O_n$, the synthesis step refines M_h^A in order to produce the behavioral protocol M of the concrete mediator for $P_1, \ldots, P_n$. That is, according to a suitable notion of parallel composition of behavioral protocols, when put in parallel with $P_1, \ldots, P_n$, the mediator M makes them able to interoperate, i.e., to interact through the mediator itself in order to achieve at least one common application-level business goal.

Clearly, the soundness of the method unavoidably depends on the semantic correctness of the specified protocol ontologies. That is, if the protocol ontologies specify incorrect relations then the method can fail. Thus, hereon we will assume to always deal with correctly specified protocol ontologies, for which our method is correct by construction (see Theorems 1 and 2 in Sections 4.6 and 4.7, respectively).

4. Method formalization

In this section, we give the formal definition of: (i) behavioral protocol (Section 4.1), (ii) parallel composition (Section 4.2), and (iii) protocol interoperability (Section 4.3). Furthermore, we define the notion of ontology (Section 4.4) and the three steps of our method including the notions of abstract behavioral protocol, abstract mediator protocol and concrete mediator protocol (Sections 4.5, 4.6, and 4.7 respectively). In Sections 4.6 and 4.7 we prove the correctness of our method for what concerns the synthesis of the abstract and the concrete mediator, respectively. The formalization is illustrated by describing our method at work on the scenario introduced in Section 2.

4.1. Behavioral protocol

A behavioral protocol is specified as a state machine describing the allowed sequences of send and receive actions that are observable from outside. Send and receive actions model the sending and receiving of messages. Actions that cannot be observed from outside are modeled as internal actions. Note that the considered send/receive primitives are sufficient to model the behavior of higher-level transmission primitives, such as WSDL[3] request-response and one-way operations.

Messages have a type associated with them, i.e., they are typed messages. A message type is represented by a tuple of typed elements. Each typed element is a pair of a name and a type. The former is the name of the element and it is specified by means of an alphanumeric string; the latter is (the name of) the type of the element. Thus, before giving the definition of behavioral protocol (Definition 3), we need to define the notions of type, typed element, and typed message.

Let $\mathcal{D}_0$, $\mathcal{D}_1$, ..., $\mathcal{D}$, $\mathcal{D}'$, $\mathcal{D}''$, ... be distinct *domains of values*, and let t_0, t_1, ..., t, t', t'', ... be type names denoting *basic types*, one for each domain of values. For instance, $\mathcal{D}_0$ could represent the set `{true, false}` of boolean values, and t_0 could be the name `boolean` denoting the basic type of boolean values.

Out of basic types, we can recursively define the notion of type as follows.

Definition 1 *(Type).* A *type* ρ is such that either:

- $\rho = t$ for some basic type t or
- $\rho = [e1 : \rho_1; \ldots; e_m : \rho_m;]$ for some element names $e_1, \ldots, e_m$ and types $\rho_1, \ldots, \rho_m$ (i.e., $e1 : \rho_1$, ..., $e_m : \rho_m$ are typed elements).

Definition 1 allows us to deal with *complex types* obtained by hierarchically combining types in (possibly nested) tuples of typed elements, hence considering also sequences of elements of the same type. For instance, by referring to the behavioral protocol of BC shown in Fig. 2, the tuple `[orderID:string; item:Item;]` denotes the type of the message named `AddItem` where `orderID` and `item` are element names, and `string` and `Item` are a basic type and a complex type, respectively. `Item` is in turn the type `[id:string; quantity:int; descr:string;]`. Thus, within our formalization, the type of `AddItem` is represented as the tuple `[orderID:string; item:[id:string; quantity:int; descr:string;];]`.

Exploiting Definition 1, we can define the notion of typed message as follows.

Definition 2 *(Typed message).* A *typed message* is a pair (l, t) where:

- l is an alphanumeric string denoting the name of the message;
- t is the type of the message.

For instance, by continuing the above example, `AddItem!` is a typed output message whose name is `AddItem` and whose type is `[orderID:string; item:[id:string; quantity:int; descr:string;];]`.

The following definition formalizes the notion of behavioral protocol.

Definition 3 *(Behavioral protocol).* A *Behavioral Protocol* $P = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ is a state machine where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state. $\mathcal{M} = \mathcal{M}^? \cup \mathcal{M}^!$, $\mathcal{M}^?$ is a finite set of typed input messages and $\mathcal{M}^!$ is a finite set of typed output messages. $\delta \subseteq \mathcal{S} \times (\mathcal{M} \cup \{\tau\}) \times \mathcal{S}$ is the transition relation. $\mathcal{F} \subseteq \mathcal{S}$ is the set of final states. Transitions can have the following three forms:

- $(s_i, (l_1, t_1)?, s_j)$ models a *receive action*, i.e., the receiving of a typed message $(l_1, t_1) \in \mathcal{M}^?$;
- $(s_i, (l_1, t_1)!, s_j)$ models a *send action*, i.e., the sending of a typed message $(l_1, t_1) \in \mathcal{M}^!$;
- (s_i, τ, s_j) models an *internal action*.

If m is a typed message, we write $s_i \xrightarrow{m!} s_j$ to denote that $(s_i, m!, s_j) \in \delta$, and $s_i \xrightarrow{m?} s_j$ to denote that $(s_i, m?, s_j) \in \delta$. We also write $s_i \xrightarrow{\tau} s_j$ to denote that $(s_i, \tau, s_j) \in \delta$.

For a state $s \in \mathcal{S} \setminus \mathcal{F}$, we write $s \longrightarrow$ to denote that $\exists m \in \mathcal{M}, s' \in \mathcal{S} : s \xrightarrow{m!} s' \vee s \xrightarrow{m?} s' \vee s \xrightarrow{\tau} s'$. In other words, if $s \longrightarrow$ then s is not a "sink" state. Note that, in our context, sink states model possible errors in the protocol interaction due to, e.g., concurrency issues such as deadlocks.

Given a typed message $m = (l, t)$, we use the function $Elems(\cdot)$ to obtain the set of typed elements that constitute t. For instance, $Elems$((`AddItem`, `[orderID:string; item:[id:string; quantity:int; descr:string;];]`)) = {`orderID:string`, `id:string`, `quantity:int`, `descr:string`}.

[3] www.w3.org/TR/wsdl20/.

For the sake of clarity, in the graphical representation of a behavioral protocol (see Figs. 1 and 2), a transition is labeled simply with the name of the message postfixed with the send/receive flag (! or ?). The type of the message is specified on the bottom side of the figure.

A *trace* of protocol P is a sequence of actions (observable and/or internal) starting from the initial state and leading to a final state through the transitions. The set of all traces is denoted by $L(P)$ and it represents the *language* of P. With reference to Fig. 2, the sequence $StartOrder!$ $StartOrderResp?$ $AddItem!$ $AddItem!$ $Confirmation!$ $CCInfo!$ $PlaceOrder?$ $Notification?$ denotes a trace, modulo the information about the type that is omitted here for simplicity.

A protocol *relabeling* $P[f_i]$ is a behavioral protocol obtained by applying a relabeling function f_i to the protocol P. For our purposes, f_i replaces each typed message m in P with m_i. As it will be clear in Section 4.6, relabeling is a useful means to force protocols sharing complementary common actions to interact only through a mediator. In other words, while modeling the composed/mediated system, relabeling is used to suitably interpose the mediator among the considered protocols. Note that, in the practice of software development, relabeling can be easily achieved by building simple wrappers.

4.2. Parallel composition

The following definition formalizes the notion of parallel composition of an arbitrary number $n \geq 2$ of behavioral protocols $P_1, \ldots, P_n$. The state machine obtained by the parallel composition is, in turn, a behavioral protocol modeling the behavior of the system built by putting in parallel $P_1, \ldots, P_n$.

Definition 4 *(Parallel composition).* Let $P_1 = (\mathcal{S}_1, s_{01}, \mathcal{M}_1, \delta_1, \mathcal{F}_1)$, …, $P_n = (\mathcal{S}_n, s_{0n}, \mathcal{M}_n, \delta_n, \mathcal{F}_n)$ be $n \geq 2$ behavioral protocols. The *Parallel Composition* of $P_1, \ldots, P_n$ is a behavioral protocol $P = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ where $\mathcal{S} \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, $s_0 \in S$ such that $s_0 = (s_{01}, \ldots, s_{0n})$, $\mathcal{F} \subseteq \mathcal{F}_1 \times \ldots \times \mathcal{F}_n$, and $\mathcal{M} = \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$.

For $s = (s_1, \ldots, s_n) \in \mathcal{S}$ and $s' = (s'_1, \ldots, s'_n) \in \mathcal{S}$, the transition relation $\delta \subseteq \mathcal{S} \times (\mathcal{M} \cup \{\tau\}) \times \mathcal{S}$ is such that:

internal action due to synchronization on shared actions

$s \xrightarrow{\tau} s' \in \delta$ if $\exists i, j \in [1..n]$: $m \in \mathcal{M}_j^! \cap \mathcal{M}_i^?$,

$s_j \xrightarrow{m!} s'_j \in \delta_j$, $s_i \xrightarrow{m?} s'_i \in \delta_i$,

$\forall k \in [1..n] : k \neq i, j \implies s'_k = s_k$

internal action from one protocol

$s \xrightarrow{\tau} s' \in \delta$ if $\exists i \in [1..n]$: $s_i \xrightarrow{\tau} s'_i \in \delta_i$,

$\forall j \in [1..n] : j \neq i \implies s'_j = s_j$

Given n behavioral protocols $P_1, \ldots, P_n$, their parallel composition is denoted by $P_1 || \ldots || P_n$. It models the behavior of the system composed of $P_1, \ldots, P_n$. The composed system is obtained by forcing synchronization on actions shared by pairs of $P_1, \ldots, P_n$. They are complementary send/receive actions that share both the name and the type of the message.[4] At the level of the parallel composition, a synchronization produces an internal action. A τ action is produced also when one protocol performs an internal action. Finally, observable actions performed by one protocol and not shared with the other protocols are not considered, hence producing no action at the level of the parallel composition. This means that all send/receive actions are blocking and, hence, in order to progress, protocols can only either synchronize on shared actions or progress independently through the execution of internal actions. As it will be clear in the next section, this notion of parallel composition that forces synchronization on shared action is key to reason on protocol interoperability.

By referring to Figs. 1 and 2, let $MS = (\mathcal{S}_{MS}, s_0^{MS}, \mathcal{M}_{MS}, \delta_{MS}, \mathcal{F}_{MS})$ and $BC = (\mathcal{S}_{BC}, s_0^{BC}, \mathcal{M}_{BC}, \delta_{BC}, \mathcal{F}_{BC})$ be the protocols for MS and BC, respectively. $MS||BC$ results in a "no-op" behavioral protocol, meaning that it does not perform any action since its transition function is empty: $MS||BC = (\mathcal{S}_{MS} \times \mathcal{S}_{BC}, (s_0^{MS}, s_0^{BC}), \mathcal{M}_{MS} \cup \mathcal{M}_{BC}, \varnothing, \mathcal{F}_{MS} \times \mathcal{F}_{BC})$.

Note also that the set of traces of $MS||BC$ is empty, i.e., $L(MS||BC) = \varnothing$. This means that the two protocols MS and BC are not able to interoperate directly (as already discussed in Section 2).

4.3. Protocol interoperability

Leveraging the parallel composition of protocols, the following definition formalizes the notion of protocol interoperability.

Definition 5 *(Protocol interoperability).* Let $P_1, \ldots, P_n$ be $n \geq 2$ behavioral protocols and let $P_1 || \ldots || P_n = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ be the behavioral protocol of their parallel composition. $P_1, \ldots, P_n$ are *interoperable*, denoted as $P_1 | \ldots | P_n$, if and only if:

[4] We recall that, in Definition 4, m denotes a pair (l, t) of message name l and message type t.

at least one trace
$L(P_1||\ldots||P_n) \neq \varnothing$ and
no sink states
$\forall s \in \mathcal{S} : s \notin \mathcal{F} \implies s \longrightarrow$.

For a set of protocols to be interoperable, we require that they are able to interact (through messages exchange) in order to achieve at least one common application-level business goal, i.e., to reach at least one composite system's final state from the initial one. Furthermore, we require that the parallel composition of the considered protocols does not contain sink states that are not final states. In other words, $P_1|\ldots|P_n$ holds if all the interactions modeled by $P_1||\ldots||P_n$ eventually lead to final states.

Thus, if the protocols given as input to our method already interoperate, our method terminates without proceeding in performing the abstraction, matching, and synthesis steps, since a mediator for the considered protocols is not required. Otherwise, the steps overviewed in Section 3, and formalized in the following, are performed.

4.4. Ontology

Before giving the formal definition of ontology, it is worth to anticipate that, in our setting, a mediator has an input-output behavior (not necessarily strictly sequential, e.g., for allowing reordering of messages), and it is a reactive software entity that harmonizes the interaction of heterogeneous systems. A mediator "intercepts" output messages from one system, eventually issuing the *co-related* input messages to another system.

As anticipated in Sections 2 and 3, for the systems to be mediated, we assume to have the specification of their behavioral protocols enriched with protocol ontologies, which in turn refer to a domain ontology. The ontological concepts in both the protocol ontologies and the domain ontology semantically characterize typed messages. Thus, considering the way concepts of the protocol ontologies are related to concepts of the same domain ontology allows us to abstract the two protocols with respect to a "common language".

In this section, we detail how our method exploits ontological information to infer the required message co-relations while performing the abstraction, matching and synthesis steps.

Definition 6 *(Ontology).* An *ontology* O is a tuple $(C, \lhd, \rightarrow)$, where C is the finite set of *ontological concepts*, each of them represented by a typed message. For concepts in C, $\lhd \subseteq C \times C$ is the *subsumption* relation, and $\rightarrow \subseteq C \times C$ is the *inclusion* relation.

We shall write that m *is subsumed by* m', denoted by $m' - \triangleright m$, if $(m, m') \in \lhd$. That is, the set of typed elements constituting m is a subset of those constituting m', i.e., the type of m' is a subtype of the type of m.

We shall write that either all or part of m' *is included in* m, denoted by $m \rightarrow m'$, if $(m, m') \in \rightarrow$. That is, the set of typed elements constituting m includes either all the elements constituting m' or part of them, i.e., the type of m contains either entirely or partially the type of m'.

Furthermore, we shall write $m \longmapsto_O m'$ if either $m - \triangleright m' \vee m \rightarrow m'$ or $\exists k > 0 : \exists m^1, \ldots, m^k \in C : m \longmapsto_O m^1 \longmapsto_O \ldots \longmapsto_O m^k \longmapsto_O m'$. We also write $m \not\longmapsto_O$ if it does not exist any m' such that $m \longmapsto_O m'$.

The protocol ontology O_{BC} shown in Fig. 5 is the tuple $(C_{BC}, \lhd_{BC}, \rightarrow_{BC})$ where:

- $C_{BC} = \{order, item, credentials, payment, creditCard, bill, startOrder, addItem, confirmation, placeOrder, startOrderResp, ccInfo, notification\}$ where
  - $order = (Order, [id : string;])$;
  - $item = (Item, [id : string; qty : int; description : string;])$;
  - $credentials = (Credentials, [username : string; password : string])$;
  - $payment = (Payment, [ccard : [type : string; number : string; expiration : date; cvv : int; holder : string;]; bill : [text : string;];])$;
  - $creditCard = (CreditCard, [type : string; number : string; expiration : date; cvv : int; holder : string;])$;
  - $bill = (Bill, [text : string;])$; and analogously for the others remaining typed messages (their respective names and types are shown in Figs. 2 and 5);
- $\lhd_{BC} = \{confirmation - \triangleright order, placeOrder - \triangleright order, startOrderResp - \triangleright order$
  $startOrder - \triangleright credentials, ccInfo - \triangleright creditCard, notification - \triangleright bill\}$;
- $\rightarrow_{BC} = \{addItem \rightarrow item, payment \rightarrow creditCart, payment \rightarrow bill\}$.

4.5. Abstraction

The abstraction step concerns elevating the type of messages and the order they are exchanged in a behavioral protocol to a more abstract alphabet of actions that refers to concepts in the domain ontology. The following definition provides a formal characterization of the abstraction step.

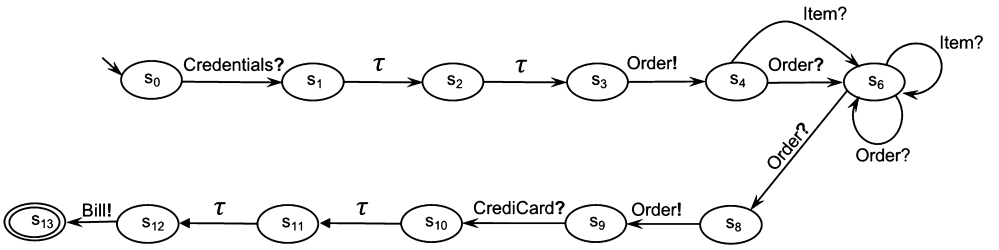

Fig. 7. Abstract behavioral protocol of MS.

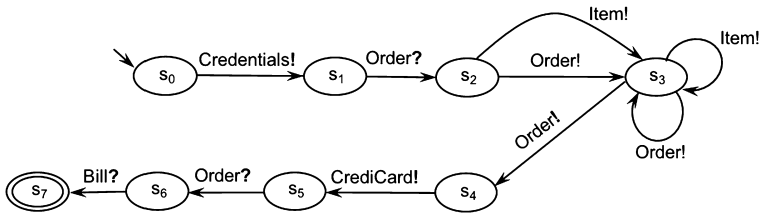

Fig. 8. Abstract behavioral protocol of BC.

Definition 7 *(Abstract behavioral protocol).* Let $P = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ be a behavioral protocol, and let $O_P = (C_P, \lhd_P, \rightarrow_P)$ be a protocol ontology for P, the corresponding *abstract behavioral protocol* $P^A = (\mathcal{S}', s_0, \mathcal{M}, \delta', \mathcal{F})$ is such that $\mathcal{S}' \subseteq \mathcal{S}$ and:

receive action

$s^0 \xrightarrow{m^k?} s^k \in \delta'$ if $\exists k > 0, s^0 \xrightarrow{m^0?} s^1, s^1 \xrightarrow{m^1?} s^2, \ldots, s^{k-1} \xrightarrow{m^{k-1}?} s^k \in \delta$:
$m^0 \longmapsto_{O_P} m^k, m^1 \longmapsto_{O_P} m^k, \ldots, m^{k-1} \longmapsto_{O_P} m^k,$
$\forall x, y \in [0..k-1] : x \neq y \implies m^x \rightarrow_P m^y$

send action

$s^0 \xrightarrow{m^k!} s^k \in \delta'$ if $\exists k > 0, s^0 \xrightarrow{m^0!} s^1, s^1 \xrightarrow{m^1!} s^2, \ldots, s^{k-1} \xrightarrow{m^{k-1}!} s^k \in \delta$:
$m^0 \longmapsto_{O_P} m^k, m^1 \longmapsto_{O_P} m^k, \ldots, m^{k-1} \longmapsto_{O_P} m^k,$
$\forall x, y \in [0..k-1] : x \neq y \implies m^x \rightarrow_P m^y$

internal action

(1) $s \xrightarrow{\tau} s' \in \delta'$ if $\exists s \xrightarrow{m!} s' \in \delta$: $m \notin C_P \vee (m \in C_P \wedge m \not\longmapsto_{O_P})$

(2) $s \xrightarrow{\tau} s' \in \delta'$ if $\exists s \xrightarrow{m?} s' \in \delta$: $m \notin C_P \vee (m \in C_P \wedge m \not\longmapsto_{O_P})$

(3) $s \xrightarrow{\tau} s' \in \delta'$ if $s \xrightarrow{\tau} s' \in \delta$

Informally, in the simplest case when $k = 1$, the message m^0 of the concrete protocol is abstracted by the message m^1, which is semantically related to m^0 directly or via a chain of ontological relations. Note that m^1 is a concept in both the protocol ontology and the domain ontology. In the general case when $k > 1$, the sequence of messages m^0, m^1, …, m^{k-1} of the concrete protocol can be abstracted by the single message m^k if the messages in the sequence are (i) all related to m^k, and (ii) mutually related one with the other through inclusion relations. In fact, in this case, they represent messages that either can be merged into one single message, by a possible mediator, or result from splitting one single message into them. Furthermore, the messages that have not been abstracted have been all replaced by τ actions. As a consequence, all the abstract protocols obtained through the abstraction step share the same alphabet of actions.

Intuitively, the abstract behavioral protocol of MS in Fig. 7 specifies that, with respect to (and at the level of) the domain ontology in Fig. 3, the allowed sequences of messages that can be sent or received by MS concern three main phases: (i) receiving of the credentials; (ii) managing the order and the adding of items to it; (iii) handling the payment through credit card and sending the bill. Note that, since `SelectItem` and `SetQuantity` are both related to `Item` and `Order`, and are mutually related one with the other, the sequence of actions *SelectItem*? *SetQuantity*? from s_4 to s_6 in Fig. 1 has been abstracted by the single action *Order*? or *Item*? from s_4 to s_6 in the figure (see the **receive action** rule). Similar considerations apply for the abstract behavioral protocol of BC shown in Fig. 8.

Abstract behavioral protocols are behavioral protocols and hence parallel composition, as per Definition 4, applies to them.

4.6. Matching

The purpose of the matching step is to build the abstract mediator M_h^A out of the abstract behavioral protocols $P_1^A, \ldots, P_n^A$. As far as reordering of messages is concerned, we recall that M_h^A can perform it through a message buffer whose size is set to a specified bound. Let h be this bound, then we write that M_h^A is h-bounded. If $L(M_h^A) = \varnothing$ there is no way to suitably mediate the abstract protocols, modulo the specified size of the buffer. This means that a concrete mediator for $P_1, \ldots, P_n$ does not exist, under the specified boundness condition. Otherwise, the abstract mediator $\overrightarrow{M_h^A}$, synthesized as a sink-free model of M_h^A, is the one that makes $P_1^A[f_1], \ldots, P_n^A[f_n]$ interoperable through the mediator itself, i.e., $P_1^A[f_1]|\ldots|P_n^A[f_n]|\overrightarrow{M_h^A}$ holds. As such, $\overrightarrow{M_h^A}$ serves as basis for synthesizing the concrete mediator.

We also recall that, due to the relabeling, all the $P_i^A[f_i]$ are decoupled since their sets of typed messages are disjoint. Within our method, this is done to force all the interactions passing through the mediator. Thus, the abstract mediator has to be synthesized by accounting for the application of the relabeling to the abstract protocols.

In the following definition, we formalize the mediator message buffer as a string, i.e., a sequence of typed messages. Let B be the mediator message buffer, $|B|$ denotes its size.

Definition 8 *(Abstract mediator protocol).* Let $P_1^A[f_1] = (\mathcal{S}_1, s_{01}, \mathcal{M}_1, \delta_1, \mathcal{F}_1)$, …, $P_n^A[f_n] = (\mathcal{S}_n, s_{0n}, \mathcal{M}_n, \delta_n, \mathcal{F}_n)$ be the *relabeled* abstract behavioral protocols for protocols $P_1, \ldots, P_n$, respectively (with $n \geq 2$). Let $h > 0$ be the bound on the size of the mediator message buffer.

The *h-bounded abstract mediator protocol* for $P_1^A[f_1], \ldots, P_n^A[f_n]$ is the behavioral protocol $M_h^A = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ where $\mathcal{S} \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \times \mathcal{M}^h$, $s_0 \in \mathcal{S}$ such that $s_0 = (s_{01}, \ldots, s_{0n}, \epsilon)$, $\mathcal{M} = \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$, $\mathcal{F} \subseteq \mathcal{F}_1 \times \ldots \times \mathcal{F}_n \times \mathcal{M}^h$, and ϵ is the empty string denoting the empty buffer.

For $s = (s_1, \ldots, s_n, B) \in \mathcal{S}$ and $s' = (s'_1, \ldots, s'_n, B') \in \mathcal{S}$, the transition relation $\delta \subseteq \mathcal{S} \times (\mathcal{M} \cup \{\tau\}) \times \mathcal{S}$ is such that:

receive action
$s \xrightarrow{m_i?} s' \in \delta$ if $\exists i, j \in [1..n], i \neq j : m_i \in \mathcal{M}_i^!, m_j \in \mathcal{M}_j^?, s_i \xrightarrow{m_i!} s'_i \in \delta_i$,
$\forall k \in [1..n] : k \neq i \implies s'_k = s_k$, $|B| < h$ and $B' = Bm_i$

send action
$s \xrightarrow{m_i!} s' \in \delta$ if $\exists i, j \in [1..n], i \neq j : m_i \in \mathcal{M}_i^?, m_j \in \mathcal{M}_j^!, s_i \xrightarrow{m_i?} s'_i \in \delta_i$,
$\forall k \in [1..n] : k \neq i \implies s'_k = s_k$,
$B' = HH'$, and $B = Hm_qH'$ with $H \in (\mathcal{M} \setminus \{m_q\})^*$, $H' \in \mathcal{M}^*$,
$q \in [1..n]$

internal action
$s \xrightarrow{\tau} s' \in \delta$ if $\exists i \in [1..n] : s_i \xrightarrow{\tau} s'_i \in \delta_i$,
$\forall j \in [1..n] : j \neq i \implies s'_j = s_j$, and $B' = B$

By considering that all the P_i^A share the same alphabet of actions, the abstract mediator receives a message m_i from a protocol $P_i^A[f_i]$ and puts it into the buffer (see the **receive action** rule). According to the specified bound, this is done only if the buffer is not full.

Whenever the mediator can synchronize with a protocol $P_i^A[f_i]$ that expects to receive m_i, if at least one occurrence of m_q is in the buffer (for some $q \in [1..n]$), then an instance of m_i built out of m_q is sent to $P_i^A[f_i]$ and m_q is removed from the buffer (see the **send action** rule). In other words, while accounting for the protocol relabeling, the abstract mediator sends a message to a certain protocol only if it has previously received – via messages exchange – the data sufficient for the production of the message.

Finally, when a protocol performs an internal action, the mediator performs a τ action as well (see the **internal action** rule).

Let P_1^A and P_2^A be the abstract protocols of MS and BC as shown in Figs. 7 and 8, respectively. The bottom side of Fig. 9 shows the abstract mediator M_2^A for $P_1^A[f_1]$ and $P_2^A[f_2]$, whose protocols are reported in the top side of the figure and in the middle, respectively. The buffer is 2-bounded and its content is delimited by '[' and ']'.

It is worth to note that, from s_8, the abstract protocol $P_1^A[f_1]$ performs the sequence of actions $Order_1!$ $CreditCard_1?$ whereas $P_2^A[f_2]$ does exactly the opposite, see the sequence $CreditCard_2!$ $Order_2?$ from s_4. M_2^A solves this mismatch by using the buffer to perform the needed reordering of messages, see the sequence of actions from s_{22} to s_{33} and the various buffer configurations in the different intermediate states. A further aspect to note is that, by using a bounded buffer, the mediator prevents possibly unbounded behavior originating from the two looping output actions in the state s_3 of $P_2^A[f_2]$.

Referring to our notion of interoperability as per Definition 5, our method aims to produce an abstract mediator that prevents the achievement of sink states that are not final states. Thus, as formalized by the following definition, the behavior protocol of the abstract mediator that is finally produced by our method is the one where all the sequences of transitions leading to non-final sink states have been suitably pruned. Still according to our notion of interoperability, this further step

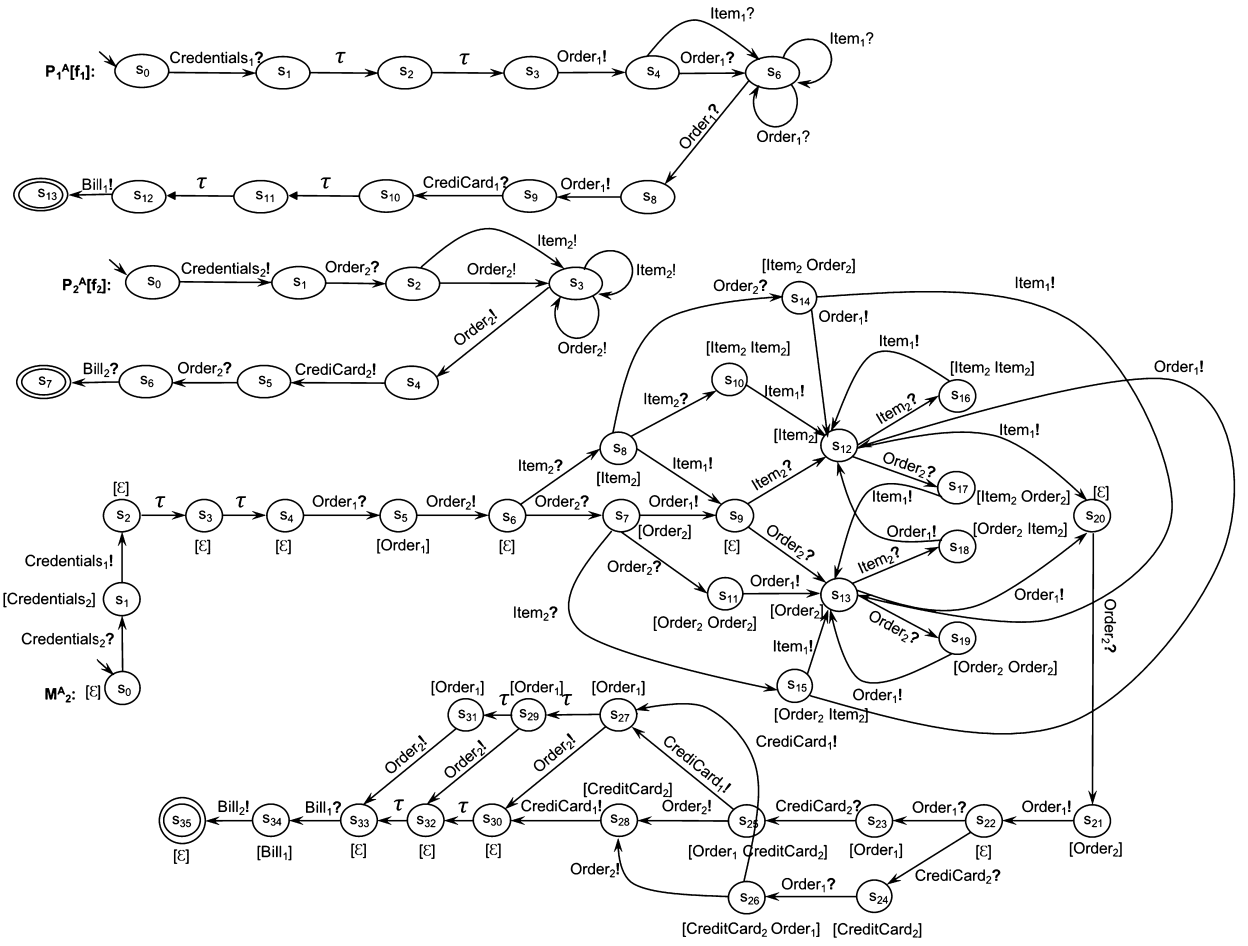

Fig. 9. 2-bounded abstract mediator for the abstract protocols of MS and BC.

is performed only if the language of the mediator protocol is not empty. Otherwise, our method can terminate since no mediator exists for the considered protocols and bound.

Definition 9 *(Sink-free abstract mediator protocol).* Let $M_h^A = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ be the h-bounded abstract mediator protocol for the abstract protocols $P_1^A[f_1], \ldots, P_n^A[f_n]$ (with $n \geq 2$), such that $L(M_h^A) \neq \varnothing$. The *sink-free abstract mediator protocol* of M_h^A is the behavioral protocol $\overrightarrow{M_h^A} = (\mathcal{S}', s_0, \mathcal{M}', \delta', \mathcal{F})$ where $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{M}' \subseteq \mathcal{M}$, $\delta' \subseteq \delta$, and $\forall s \in \mathcal{S}' : s \notin \mathcal{F} \implies s \longrightarrow$.

Note that given the protocol of the abstract mediator, its sink-free protocol is not unique. For the purposes of the work described in this paper, our method generates the maximal sink-free abstract mediator protocol with respect to the cardinality of δ, i.e., it derives the most permissive sink-free mediator. This is done by performing a slightly revised version of our previous algorithm for ensuring deadlock-freedom of protocols [36], i.e., sink-freedom in our context.

For the purposes of this paper it is sufficient to say that if our method outputs the abstract mediator protocol then: (i) its language is not empty, (ii) it satisfies the conditions formalized by Definition 9, and (iii) it is the maximal one.

Continuing our scenario, the sink-free abstract mediator protocol $\overrightarrow{M_h^A}$ is equal to the 2-bounded abstract mediator protocol M_2^A shown in Fig. 9 since the latter does not contain non-final sink states.

By exploiting Definition 5, it easy to check that $P_1^A[f_1] | P_2^A[f_2] | \overrightarrow{M_h^A}$ holds and, hence, as stated by the following theorem, the abstract mediator for our illustrative scenario is correct by construction. That is, if there exists at least one trace in the language of the mediator protocol, then the sink-free abstract mediator makes the abstract protocols interoperable (through the mediator itself).

Theorem 1 *(Correctness of the abstract mediator). Let $P_1, \ldots, P_n$ be $n \geq 2$ behavioral protocols with respective ontologies $O_1, \ldots, O_n$ all referring to the same domain ontology. Let $P_i^A = (\mathcal{S}_i, s_i^0, \mathcal{M}_i, \delta_i, \mathcal{F}_i)$ be the abstract protocol obtained out of P_i and O_i ($i \in [1..n]$).*

Let M_h^A be the h-bounded abstract mediator protocol for $P_1^A[f_1], \ldots, P_n^A[f_n]$ such that $L(M_h^A) \neq \varnothing$. Let $\overrightarrow{M_h^A} = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ be the sink-free abstract mediator protocol of M_h^A, the following property holds:

$$P_1^A[f_1] \mid \ldots \mid P_n^A[f_n] \mid \overrightarrow{M_h^A}.$$

Proof. By contradiction, let us suppose that $L(\overrightarrow{M_h^A}) \neq \varnothing$ and $P_1^A[f_1] \mid \ldots \mid P_n^A[f_n] \mid \overrightarrow{M_h^A}$ does not hold, i.e., the protocols plus the mediator does not interoperate. That is, $L(\overrightarrow{M_h^A}) \neq \varnothing$ and **(i)** $L(P_1^A[f_1] \,||\ldots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A}) = \varnothing$ or **(ii)** $P_1^A[f_1] \,||\ldots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A}$ is not sink-free.

Case (i). If $L(\overrightarrow{M_h^A}) \neq \varnothing$ then $\exists r > 0, t \in L(\overrightarrow{M_h^A}) : t = \alpha_1 \ldots \alpha_r$.

In turn, $\exists r > 0 \;:\; \exists (s_1^0, \ldots, s_n^0, B^0), (s_1^1, \ldots, s_n^1, B^1), \ldots, (s_1^r, \ldots, s_n^r, B^r) \in \mathcal{S}$: $(s_1^0, \ldots, s_n^0, B^0) \xrightarrow{\alpha_1} (s_1^1, \ldots, s_n^1, B^1) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_r} (s_1^r, \ldots, s_n^r, B^r)$, with $(s_1^0, \ldots, s_n^0, B^0)$ being the initial state s_0 of $\overrightarrow{M_h^A}$ and $B^0 = \epsilon$ its initial empty buffer.

By Definition 8, $\exists i \in [1..n] \;:\; \exists m_i \in \mathcal{M} \;:\; \alpha_k = m_i? \vee \alpha_k = m_i! \vee \alpha_k = \tau$ for all $k \in [1..r]$. Let δ_{MS} be the transition function of the mediated system $(P_1^A[f_1] \,||\ldots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A})$. By Definitions 4 and 8, and by the relabeling:

- if $\alpha_k = m_i?$ for some $m_i \in \mathcal{M}^?$ and $|B| < h$ then
  $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, B^{k-1})) \xrightarrow{\tau}$
  $(s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, B^{k-1} m_i)) \in \delta_{MS}$,
  due to the synchronization of $m_i?$ and $m_i!$ performed by $\overrightarrow{M_h^A}$ and $P_i^A[f_i]$, respectively;
- if $\alpha_k = m_i!$ for some $m_i \in \mathcal{M}^!$ then
  $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, H m_q H')) \xrightarrow{\tau}$
  $(s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, HH')) \in \delta_{MS}$,
  due to the synchronization of $m_i!$ and $m_i?$ performed by M_h^A and $P_i^A[f_i]$, respectively, with $H \in (\mathcal{M} \setminus \{m_q\})^*$, $H' \in \mathcal{M}^*$, $q \in [1..n]$;
- if $\alpha_k = \tau$ then
  $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, B^{k-1})) \xrightarrow{\tau}$
  $(s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, B^{k-1})) \in \delta_{MS}$,
  due to the τ action performed by $P_i^A[f_i]$;

Case (ii). If $\overrightarrow{M_h^A}$ is sink-free then $\forall s \in \mathcal{S} : s \notin \mathcal{F} \implies s \longrightarrow$. That is, by Definition 8, $\forall (s_1, \ldots, s_n, B) \in \mathcal{S} \;:\; (s_1, \ldots, s_n, B) \notin \mathcal{F} \implies \exists i \in [1..n] \;:\; \exists m_i \in \mathcal{M}, (s'_1, \ldots, s'_n, B') \in \mathcal{S} \;:\; (s_1, \ldots, s_n, B) \xrightarrow{m_i?} (s'_1, \ldots, s'_n, B') \vee (s_1, \ldots, s_n, B) \xrightarrow{m_i!} (s'_1, \ldots, s'_n, B') \vee (s_1, \ldots, s_n, B) \xrightarrow{\tau} (s'_1, \ldots, s'_n, B')$.

Let δ_{MS} be the transition function of the mediated system $(P_1^A[f_1] \,||\ldots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A})$. By Definitions 4 and 8, and by the relabeling:

- if $(s_1, \ldots, s_n, B) \xrightarrow{m_i?} (s'_1, \ldots, s'_n, B')$ for some $m_i \in \mathcal{M}^?$ and $|B| < h$ then
  $(s_1, \ldots, s_n, (s_1, \ldots, s_n, B)) \xrightarrow{\tau} (s'_1, \ldots, s'_n, (s'_1, \ldots, s'_n, B')) \in \delta_{MS}$,
  with $B' = B m_i$ and $\forall k \in [1..n] : k \neq i \implies s'_k = s_k$,
  due to the synchronization of $m_i?$ and $m_i!$ performed by $\overrightarrow{M_h^A}$ and $P_i^A[f_i]$, respectively;
- if $(s_1, \ldots, s_n, B) \xrightarrow{m_i!} (s'_1, \ldots, s'_n, B')$ for some $m_i \in \mathcal{M}^!$ then
  $(s_1, \ldots, s_n, (s_1, \ldots, s_n, B)) \xrightarrow{\tau} (s'_1, \ldots, s'_n, (s'_1, \ldots, s'_n, B')) \in \delta_{MS}$,
  with $B = H m_q H'$, $B' = HH'$ and $\forall k \in [1..n] : k \neq i \implies s'_k = s_k$,
  due to the synchronization of $m_i!$ and $m_i?$ performed by M_h^A and $P_i^A[f_i]$, respectively, with $H \in (\mathcal{M} \setminus \{m_q\})^*$, $H' \in \mathcal{M}^*$, $q \in [1..n]$;
- if $(s_1, \ldots, s_n, B) \xrightarrow{\tau} (s'_1, \ldots, s'_n, B')$ then
  $(s_1, \ldots, s_n, (s_1, \ldots, s_n, B)) \xrightarrow{\tau} (s'_1, \ldots, s'_n, (s'_1, \ldots, s'_n, B')) \in \delta_{MS}$,
  due to the τ action performed by $P_i^A[f_i]$;

Thus also all states of $(P_1^A[f_1] \,||\ldots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A})$, that are not final, are not sink states.

We have shown that the existence of $t = \alpha_1 \dots \alpha_r$ in $L(\overrightarrow{M_h^A})$ implies that there exists $t' = \tau^r$ in $L(P_1^A[f_1] \,||\dots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A})$. Furthermore, the sink freedom of $\overrightarrow{M_h^A}$ implies that also the mediated system $P_1^A[f_1] \,||\dots||\, P_n^A[f_n] \,||\overrightarrow{M_h^A}$ is sink-free. That is, the existence of at least one trace in $L(\overrightarrow{M_h^A})$ and the sink freedom of $\overrightarrow{M_h^A}$ imply that the protocol plus the mediator interoperate. This is a contradiction and, hence, the proof is given. □

4.7. Synthesis

The synthesis step aims to build the concrete mediator M out of the behavioral protocol $\overrightarrow{M_h^A}$ of the abstract one, the protocols $P_1, \dots, P_n$ to be mediated and their protocol ontologies $O_1, \dots, O_n$, all referring to the domain ontology O. We recall that if $\overrightarrow{M_h^A}$ is not empty and sink-free then M exists. The kinds of communication and coordination mismatches that M is able to solve are characterized by the connector algebra described in [37]. It is an algebra for reasoning about protocol mismatches where basic mismatches can be solved by suitably defined primitives, while complex mismatches can be settled by composition operators that build connectors out of simpler ones.

In particular, as it is shown below by means of our illustrative scenario, our method synthesizes M in such a way that it is able to perform reordering of messages according to what $\overrightarrow{M_h^A}$ already does. Furthermore, it is able to translate syntactically different sequences of messages that have the same semantics in terms of the high-level business task they allow to achieve. Finally, M is able to either split one message from one protocol into a sequence of messages to be sent to a different protocol or merge a sequence of messages into one message.

Analogously to what is done for the abstract mediator, the synthesis of M accounts for protocol relabeling. That is, M is synthesized such that $P_1[f_1]| \dots |\, P_n[f_n]|M$ holds. As it is evident from Definition 10, it is worth to mention that M is sink-free by construction due to the fact that it is synthesized as a refinement of $\overrightarrow{M_h^A}$ that is sink-free as well. Thus, in the following, in order to prove that $P_1[f_1]| \dots |\, P_n[f_n]|M$ holds, it is sufficient to prove that $L(P_1[f_1]|| \dots ||\, P_n[f_n]||M) \neq \varnothing$ (see Theorem 2).

In Definition 10, we formalize the concrete mediator as a behavioral protocol accounting for a notion of *knowledge*. The mediator knowledge K, that can vary from state to state, is encoded as a set of typed messages. At the beginning it is empty. Whenever a message is received by the mediator, it is added to K hence enhancing the mediator knowledge. For messages that are not exchanged with third-party protocols,[5] the mediator sends a message to a protocol (or even a sequence of messages in case of split) only if K contains a set of messages whose typed elements can be used to produce the message (or messages) to be sent. In particular, in Definition 10, if K contains a set of messages whose typed elements can be used to produce a message m_i to be sent to $P_i[f_i]$, then we write $K \rightsquigarrow m_i$.

More precisely, let n be the number of protocols to be mediated and, for all $k \in [1..n]$, let $O_k = (C_k, \lhd_k, \rightarrow_k)$ be their protocol ontologies referring to the domain ontology $O = (C, \lhd, \rightarrow)$. We write that $K \rightsquigarrow m_i$ if:

$\exists h > 0,\ j1, \dots, jh \in [1..n],\ m^1_{j1}, \dots, m^h_{jh} \in K,\ m' \in C \cup C_i$:

$$Elems(m_i) \subseteq Elems(m^1_{j1}) \cup \dots \cup Elems(m^h_{jh}) \qquad \text{(syntactic match)}$$
$$m \longmapsto_{O_i} m',\ m^1 \longmapsto_{O_{j1}} m',\ \dots,\ m^h \longmapsto_{O_{jh}} m' \qquad \text{(semantic match)}$$

Intuitively, according to the matching conditions above, the typed message m_i can be built from K if:

syntactic match – K contains $h > 0$ messages (received by the relabeled behavioral protocols $P_{j1}[f_{j1}], \dots, P_{jh}[f_{jh}]$) whose typed elements have the same type of the typed elements of m_i (to be sent to $P_i[f_i]$), although they could be named differently;
semantic match – these typed elements semantically match the ones of m_i since both sets of typed elements (the ones of m_i and the ones contained in K) are abstracted by the same domain ontology concept $m' \in C$ which is contained in the protocol ontology O_i of $P_i[f_i]$ (i.e., $m' \in C_i$).

For instance, by referring to our illustrative scenario, $K \rightsquigarrow Notification$ if $ConfirmPayment \in K$. In fact, the two messages syntactically match since the set of typed elements of $ConfirmPayment$ is $\{msg : string\}$ and the one of $Notification$ is $\{ack : string\}$. Furthermore, they semantically match since *Bill* is subsumed by both $Confirmation$ and $Notification$ in the respective protocol ontologies (Figs. 4 and 5).

The only exception that the method admits, concerns the handling of messages that a protocol should receive from a third-party protocol. For instance, referring to our scenario, MS expects to interact with a third-party payment system by sending the message $ThirdParyPayment$ (to perform the payment) and receiving the message $PaymentResp$ (the notification of the payment). Clearly, the mediator cannot have the knowledge to correctly instantiate the content of the $PaymentResp$ message. In fact, this has to be done by invoking some operation of the third-party payment system. Thus,

[5] That is, protocols that are not in the set of the considered ones.

in this case, our method produces the corresponding sending transition without accounting for the data contained in the knowledge. At the level of the mediator protocol, that models only the externally observable interaction with the other protocols, this is correct. At the level of the mediator actual code, as discussed later on in this section, the manual intervention of the user (e.g., the mediator developer) is required, unless the user re-runs the method by considering also the behavioral protocol of the third-party payment system.

Definition 10 *(Concrete mediator protocol).* For all $i \in [1..n]$, let $P_i[f_i] = (\mathcal{S}_i, s_{0i}, \mathcal{M}_i, \delta_i, \mathcal{F}_i)$ be the *relabeled* abstract behavioral protocol for protocol P_i and $O_i = (C_i, \lhd_i, \rightarrow_i)$ be its protocol ontology referring to the domain ontology $O = (C, \lhd, \rightarrow)$. Let $\overrightarrow{M_h^A} = (\mathcal{S}_A, s_{0A}, \mathcal{M}_A, \delta_A, \mathcal{F}_A)$, with $h > 0$, be the sink-free abstract mediator protocol for all the $P_i^A[f_i]$.

The *concrete mediator protocol* for $P_1[f_1], \ldots, P_n[f_n]$ is the behavioral protocol $M = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ where $\mathcal{S} \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \times \mathcal{S}_A \times K$ such that $K \subseteq \mathcal{M}$, $s_0 \in \mathcal{S}$ such that $s_0 = (s_{01}, \ldots, s_{0n}, s_{0A}, \varnothing)$, $\mathcal{M} = \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$, $\mathcal{F} \subseteq \mathcal{F}_1 \times \ldots \times \mathcal{F}_n \times \mathcal{F}_A \times K$.

For $s^0 = (s_1^0, \ldots, s_n^0, s_A^0, K^0)$, $s^1 = (s_1^1, \ldots, s_n^1, s_A^1, K^1)$, …, $s^y = (s_1^y, \ldots, s_n^y, s_A^y, K^y) \in \mathcal{S}$, with $y > 0$, the transition relation $\delta \subseteq \mathcal{S} \times (\mathcal{M} \cup \{\tau\}) \times \mathcal{S}$ is such that:

receive action 1

$s^0 \xrightarrow{m_i^0?} s^1, \ldots, s^{y-1} \xrightarrow{m_i^{y-1}?} s^y \in \delta$ if $\exists i \in [1..n], m_i' \in \mathcal{M}_A^?$:

$s_A^0 \xrightarrow{m_i'?} s_A^y \in \delta_A$, $s_i^0 \xrightarrow{m_i^0!} s_i^1, \ldots, s_i^{y-1} \xrightarrow{m_i^{y-1}!} s_i^y \in \delta_i$,

$m^0 \longmapsto_{O_i} m'$, $m^1 \longmapsto_{O_i} m'$, …, $m^{y-1} \longmapsto_{O_i} m'$,

$\forall x, z \in [0..y-1] : x \neq z \implies m^x \rightarrow_i m^z$,

$K^1 = K^0 \cup \{m_i^0\}$, …, $K^y = K^{y-1} \cup \{m_i^{y-1}\}$,

$s_A^{y-1} = \ldots = s_A^1 = s_A^0$, $\forall j \in [1..n] : j \neq i \implies s_j^y = \ldots = s_j^1 = s_j^0$

send action 1

$s^0 \xrightarrow{m_i^0!} s^1, \ldots, s^{y-1} \xrightarrow{m_i^{y-1}!} s^y \in \delta$ if $\exists i \in [1..n], m_i' \in \mathcal{M}_A^!$:

$s_A^0 \xrightarrow{m_i'!} s_A^y \in \delta_A$, $s_i^0 \xrightarrow{m_i^0?} s_i^1, \ldots, s_i^{y-1} \xrightarrow{m_i^{y-1}?} s_i^y \in \delta_i$,

$m^0 \longmapsto_{O_i} m'$, $m^1 \longmapsto_{O_i} m'$, …, $m^{y-1} \longmapsto_{O_i} m'$,

$\forall x, z \in [0..y-1] : x \neq z \implies m^x \rightarrow_i m^z$,

$K^0 \rightsquigarrow m_i^0, \ldots, K^0 \rightsquigarrow m_i^{y-1}$, $K^y = \ldots = K^1 = K^0$,

$s_A^{y-1} = \ldots = s_A^1 = s_A^0$, $\forall j \in [1..n] : j \neq i \implies s_j^y = \ldots = s_j^1 = s_j^0$

For $s = (s_1, \ldots, s_n, s_A, K)$, $s' = (s_1', \ldots, s_n', s_A', K')$, the transition relation $\delta \subseteq \mathcal{S} \times (\mathcal{M} \cup \{\tau\}) \times \mathcal{S}$ is such that:

receive action 2

$s \xrightarrow{m_i?} s' \in \delta$ if $s_A \xrightarrow{\tau} s_A' \in \delta_A$, $\exists i \in [1..n], m_i \in \mathcal{M}_i^!$:

$s_i \xrightarrow{m_i!} s_i' \in \delta_i$, $m \notin C_i$, $K' = K \cup \{m_i\}$,

$\forall j \in [1..n] : j \neq i \implies s_j' = s_j$

send action 2

$s \xrightarrow{m_i!} s' \in \delta$ if $s_A \xrightarrow{\tau} s_A' \in \delta_A$,

$\exists i \in [1..n], m_i \in \mathcal{M}_i^? : s_i \xrightarrow{m_i?} s_i' \in \delta_i$, $m \notin C_i$,

$\forall j \in [1..n] : j \neq i \implies s_j' = s_j$, $K' = K$

internal action

$s \xrightarrow{\tau} s' \in \delta$ if $s_A \xrightarrow{\tau} s_A' \in \delta_A$, $\exists i \in [1..n] : s_i \xrightarrow{\tau} s_i' \in \delta_i$,

$\forall j \in [1..n] : j \neq i \implies s_j' = s_j$, $K' = K$

The concrete mediator receives a sequence of messages $m_i^0, m_i^1, \ldots, m_i^{y-1}$ (for some $y > 0$) from a protocol $P_i[f_i]$ and adds them to the knowledge. According to rule **receive action 1**, this is done only if the abstract mediator receives a message m_i' that is an abstraction of the sequence of messages $m_i^0, m_i^1, \ldots, m_i^{y-1}$ (see Section 4.5). Otherwise, according to rule **receive action 2**, the mediator receives a message m_i from a protocol P_i and enhances its knowledge if m (i.e., m_i before the relabeling) is not abstracted in the protocol ontology. Note that, according to Definition 8, receiving/sending m_i resulted in a τ action in the abstract mediator. This means that m_i is a message that should be used to invoke some operation of a third-party system.

Whenever the mediator can synchronize with a protocol $P_i[f_i]$ that expects to receive a sequence of messages $m_i^0, m_i^1, \ldots, m_i^{y-1}$, if the typed elements of the messages contained in the knowledge K allow to build the messages in the sequence, they are sent to P_i. According to **send action 1**, this is done only if the abstract mediator sends a message m_i' that is an

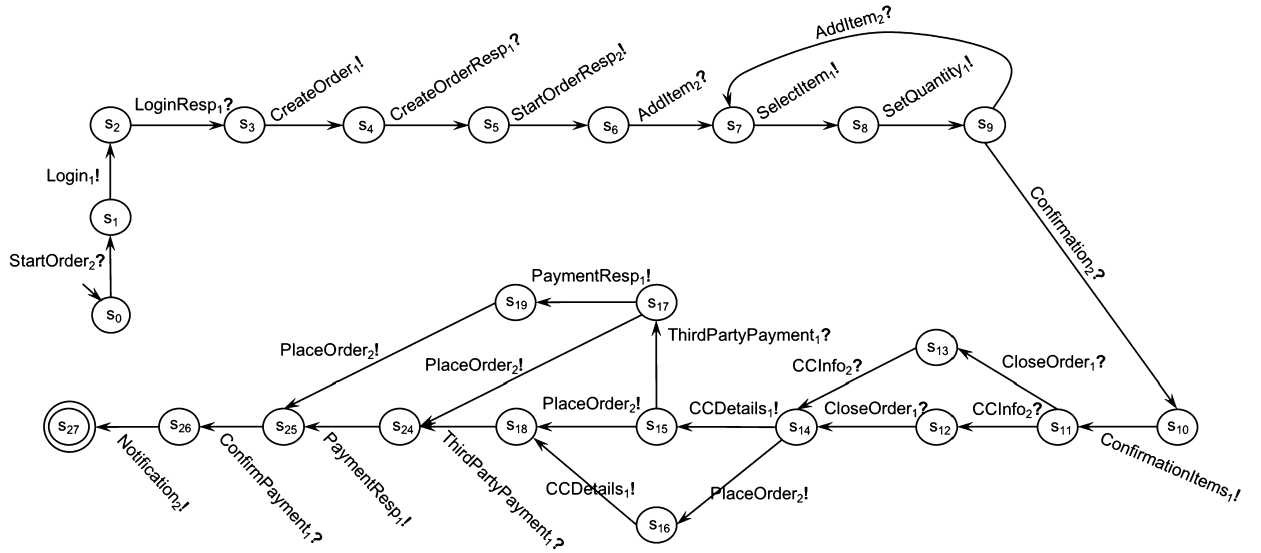

Fig. 10. Concrete mediator for MS and BC.

abstraction of the sequence of messages m_i^0, m_i^1, ..., m_i^{y-1}. Otherwise, according to **send action 2**, the mediator sends a message m_i to $P_i[f_i]$ if m (i.e., m_i before the relabeling) is not abstracted in the protocol ontology. At the level of the mediator actual code, this means that m_i is a message that has been received from a third-party system. The interaction with the third-party system has to be embedded in the generated mediator code.

When a protocol performs an internal action, according to what the abstract mediator does, the mediator performs a τ action as well (**internal action**).

In other words, similarly to the abstract mediator, the concrete mediator sends messages to a certain protocol only if it has received the data sufficient for their production – either via messages exchange with the considered protocols or thorough the aptly coded interaction with some third-party system.

Let P_1 and P_2 be the protocols of MS and BC as shown in Figs. 1 and 2, respectively. Fig. 10 shows the concrete mediator M for $P_1[f_1]$ and $P_2[f_2]$. For the sake of clarity, the internal structure of the states is not represented.

Following the formal characterization of the interoperability mismatches described in [37], beyond being capable of translating single syntactically different messages that are semantically equivalent, M is able to match the sequence of messages $Login_1?$ $LoginResp_1!$ $CreateOrder_1?$ $CreateOrderResp_1!$ with the sequence $StartOrder_2!$ $StartOrderResp_2?$ performed by $P_1[f_1]$ and $P_2[f_2]$, respectively. This is a translation of syntactically different, yet semantically equivalent, sequences of messages. In particular, the translation is obtained by consuming the extra message $LoginResp_1$ and producing the missing message $CreateOrder_1$. Here, the production does not require the interaction with a third-party system since the knowledge contains the data needed to produce the message. Furthermore, M is able to split the message $AddItem_2!$ sent by $P_2[f_2]$ into the sequence $SelectItem_1?$ $SetQuantity_1?$ received by $P_1[f_1]$. Finally, according to what $\overrightarrow{M_h^A}$ already does, M is able to reorder the sequence $CloseOrder_1!$ $CCDetails_1?$ into $CCInfo_2!$ $PlaceOrder_1?$.

For what concerns the messages $ThirdPartyPayment_1!$ and $PaymentResp_1?$ that $P_1[f_1]$ expects to exchange with a third-party protocol (different from $P_2[f_2]$), the mediator protocol is synthesized to simply receive the former and send a "void" instance of the latter. This means that the logic needed to suitably interact with the third-party payment system in order to obtain a semantically correct instance of $PaymentResp$ has to be embedded in the generated mediator code. As already anticipated, note that an alternative would be to run again the method on three protocols: P_1, P_2, and the protocol of the considered payment system (enriched with its protocol ontology).

By exploiting Definition 4, it easy to check that $L(P_1^A[f_1]||P_2^A[f_2]||M) \neq \varnothing$ and, hence, as stated by the following theorem, the concrete mediator for our illustrative scenario is correct by construction. That is, it makes the considered protocols interoperable.

Theorem 2 *(Correctness of the concrete mediator). Let $P_1, \ldots, P_n$ be $n \geq 2$ behavioral protocols with respective ontologies $O_1 = (C_1, \lhd_1, \rightarrow_1), \ldots, O_n = (C_n, \lhd_n, \rightarrow_n)$ all referring to the same domain ontology $O = (C, \lhd, \rightarrow)$. Let $\overrightarrow{M_h^A} = (\mathcal{S}_A, s_{0A}, \mathcal{M}_A, \delta_A, \mathcal{F}_A)$, with $h > 0$, be the sink-free abstract mediator for $P_1^A[f_1], \ldots, P_n^A[f_n]$ such that $L(\overrightarrow{M_h^A}) \neq \varnothing$. Let $M = (\mathcal{S}, s_0, \mathcal{M}, \delta, \mathcal{F})$ be the concrete mediator protocol for $P_1[f_1], \ldots, P_n[f_n]$, the following property holds:*

$$P_1[f_1] \,|\, \ldots \,|\, P_n[f_n] \,|M.$$

Proof. By Definition 10, M is sink-free by construction due to the fact that it is synthesized as a refinement of $\overrightarrow{M_h^A}$ that is sink-free as well. Thus, what remains to be proven is that $L(P_1[f_1] \,||\ldots||\, P_n[f_n] \,||M) \neq \varnothing$.

By hypothesis $L(M) \neq \varnothing$, otherwise $L(\overrightarrow{M_h^A}) \neq \varnothing$ would not be true. By contradiction, let us suppose that $L(P_1[f_1] \,||\ldots||\, P_n[f_n] \,||M) = \varnothing$. This means that $\exists r > 0, t \in L(M) : t = \alpha_1 \ldots \alpha_r$.

In turn, $\exists r > 0 \;:\; \exists(s_1^0, \ldots, s_n^0, s_A^0, K^0),\ (s_1^1, \ldots, s_n^1, s_A^1, K^1),\ \ldots,\ (s_1^r, \ldots, s_n^r, s_A^r,\ K^r) \in \mathcal{S} \;:\; (s_1^0, \ldots, s_n^0, s_A^0, K^0) \xrightarrow{\alpha_1} (s_1^1, \ldots, s_n^1, s_A^1, K^1) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_r} (s_1^r, \ldots, s_n^r, s_A^r,\ K^r)$, with $(s_1^0, \ldots, s_n^0, s_A^0, K^0)$ being the initial state s_0 of M and $K^0 = \varnothing$ its initial empty knowledge.

By Definition 10, $\exists i \in [1..n] \;:\; \exists m_i \in \mathcal{M} \;:\; \alpha_k = m_i? \vee \alpha_k = m_i! \vee \alpha_k = \tau$ for all $k \in [1..r]$. Let δ_{MS} be the transition function of the mediated system $(P_1[f_1] \,||\ldots||\, P_n[f_n] \,||M)$. By Definitions 4 and 10, and by the relabeling:

- if $\alpha_k = m_i?$ for some $m_i \in \mathcal{M}^?$ then
  (1) $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, s_A^{k-1}, K^{k-1}))$
  $\xrightarrow{\tau} (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, s_A^{k}, K^{k-1} \cup \{m_i\}))$
  $\in \delta_{MS}$, due to the synchronization of $m_i?$ and $m_i!$ performed by M and $P_i[f_i]$, respectively, and to the action $m_i'?$ performed by M_h^A, with $m \longmapsto_{O_i} m'$ and $m' \in C_i \cap C$;
  (2) $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, s_A^{k-1}, K^{k-1}))$
  $\xrightarrow{\tau} (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, s_A^{k}, K^{k-1} \cup \{m_i\}))$
  $\in \delta_{MS}$, due to the action $m_i!$ performed by $P_i[f_i]$, with $m \notin C_i$, and to the action τ performed by M_h^A;
- if $\alpha_k = m_i!$ then
  (1) $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, s_A^{k-1}, K^{k-1}))$
  $\xrightarrow{\tau} (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, s_A^{k}, K^{k-1}))$
  $\in \delta_{MS}$, due to the synchronization of $m_i!$ and $m_i?$ performed by M and $P_i[f_i]$, respectively, with $K \rightsquigarrow m_i$, and to the action $m_i'!$ performed by M_h^A, with $m \longmapsto_{O_i} m'$ and $m' \in C_i \cap C$;
  (2) $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, s_A^{k-1}, K^{k-1}))$
  $\xrightarrow{\tau} (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, s_A^{k}, K^{k-1})) \in \delta_{MS}$,
  due to the action $m_i?$ performed by $P_i[f_i]$, with $m \notin C_i$, and to the action τ performed by M_h^A;
- if $\alpha_k = \tau$ then
  $(s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k-1}, \ldots, s_n^{k-1}, s_A^{k-1}, K^{k-1}))$
  $\xrightarrow{\tau} (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, (s_1^{k-1}, \ldots, s_i^{k}, \ldots, s_n^{k-1}, s_A^{k}, K^{k-1})) \in \delta_{MS}$,
  due to the τ action performed by both $P_i[f_i]$ and M_h^A;

Thus, in any case, the existence of $t = \alpha_1 \ldots \alpha_r$ in $L(M)$, for some $r > 0$, implies that there exists $t' = \tau^r$ in $L(P_1[f_1] \,||\ldots||\, P_n[f_n] \,||M)$. This is a contradiction and, hence, the proof is given. □

5. Related work

The interoperability of protocols has received attention since the early days of networking and many efforts have been done in many contexts including for example formal approaches for protocol conversion, like in [4,5]. Protocol interoperability is a multifaceted problem to tackle including the ability to correctly coordinate and semantically mediate components or services' interaction. In the following we discuss the related works closest to ours.

Knowledge Representation, Reasoning, and Ontologies. Knowledge Representation and Reasoning have been the focus of tremendous work within Artificial Intelligence since more than twenty years [38–41]. Among largely adopted means to conceptualize knowledge there are, e.g., ontologies [42,43] and description logic [44]. Both of them provide the basis for the automated reasoning about the knowledge through inference engines. To exploit the knowledge conceptualized into an ontology for, e.g., query answering or data translation, a basic operation to do on it is the ontology mapping/matching [45,46] that finds an alignment among concepts.

Among the many works that represent and reason about knowledge by using ontologies, the work in [12] proposes a conceptualization for knowledge acquisition from heterogeneous data sources and its representation towards automating mediators generation. The work presented in [47], instead, is a conceptualization for an architecture of such knowledge representation and reasoning.

Web Services and Semantic Web. Recently, with the emergence of web services and advocated interoperability, the research community has been studying solutions to the automatic mediation of business processes [13–16]. However, most solutions are discussed informally, making it difficult to assess their respective advantages and drawbacks.

A lot of effort has been also devoted to behavioral adaptation within the web services community. Among such works, [11] relates to our work. 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. Instead, our matching is driven by the ontology.

Another related area is web services substitution and a work that is particularly related to ours is described in [48]. The method described uses an ontology to reason about components interface mapping and the synthesis of mediators according to such mapping.

Our work also closely relates to significant effort in the Semantic Web 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 [49]. This led to elicit base patterns for process mediation together with supporting algorithms [13,50].

Formal methods and software architectures. The seminal work [8] is strictly related to the notions of mediator presented in this paper. Compared to our connector synthesis, this work does not allow to deal with ordering mismatches and different granularity of the languages (solvable by the split and merge patterns).

In [19] the authors present an approach for formally specifying connector wrappers as protocol transformations, modularizing them, and reasoning about their properties, with the aim to resolve component mismatches.

In [51], the authors use game theory 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 our methods, their method needs as input a deadlock-free specification of the requirements that should be satisfied by the adaptor, by delegating to the user the non-trivial task of specifying that.

In [52] 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. Although these formalizations supports connector modularization, automated synthesis is not treated at all hence keeping the focus only on connector design and specification.

The work described in [53] proposes and formalizes a method for the automated synthesis of modular connectors through the composition of primitive protocol mediation patterns [32] represented as terms of an algebra of connectors [37]. Modularity of the synthesized connector eases connector code maintenance and evolution. Similarly to our work, the method is organized into a protocol alignment step and a mediator synthesis step. For the former, ontological information is exploited. The adopted notion of ontology is method-specific. However, in the current practice of ontology development, one cannot expect to find a highly specific (to the considered protocols) ontology as the one formalized in [53]. The production of this ontology involves the extension of a more general existing domain ontology in the domain of interest. This extension allows the definition of two specific ontologies that represent a semantic description for the two protocols to be mediated. Thus, in order to be applied, the method in [53] requires the user to define suitable mappings between the two specific protocol ontologies.

5.1. Detailed comparison of our work with the works described in [54] and [55]

In the following, we make a detailed comparison of our work with the most closely related ones, i.e., the works described in [54] and [55]. Both of them use an ontology-based reasoning together with constraint programming for the synthesis of mediators between components. Table 1 provides a summary of the comparison.

Considered inputs. All the three works consider: a behavioral model of the protocols under analysis and an ontological model of the domain referred by actions and data on the interfaces of the protocols. The main difference about the considered input is that the work in [54] uses interface automata, the work in [55] uses FSP and LTSs, and we use LTSs extended with data types.

Inference means. All the three papers infer and use some semantic correspondences by exploiting ontologies. The main difference is that, in order to use the inferred correspondences, the work in [54] encodes them as ordering constraints of I/O data, i.e., data dependencies (precedes); the work in [55] encodes the solution of constraint programming problems as processes; instead we use usual ontological relationships in the current practice of ontology specification and reasoning.

Further assumptions. All the three works consider: synchronous interaction, the possibility to cache the data, and the possibility to have several actions' mappings. A point in favor of our work with respect to the ones in [54] and [55], is that our method relaxes the so-called "fairness assumption"[6] by accounting for a deadlock-free (i.e., sink-free in our context) notion of interoperability and of the synthesized mediator. Furthermore the works in [54] and [55] consider deterministic automata without hidden actions, whereas our method allows non-deterministic automata and hidden actions; moreover in both [54] and [55] it is mentioned that several possible action mappings are possible and they seem to use the minimal one prefix. In [55], the authors state that to keep the domain finite, actions can appear only once in a sequence in the problem formulation. Differently from our approach, this suggests that the considered automata can only have loops or cycles of length equal to 1.

Produced outputs. All the three papers synthesize a most general mediator if it exists; however, the conditions for the existence and the interoperability issues addressed by the mediator are different. In particular, the mismatches that are

[6] In the presence of branching behavior, a protocol will eventually take all the possible branches.

Table 1
Comparison of our work with the two closet related works.

Aspect	Work in [54]	Work in [55]	Our work
Inputs	behavioral model	behavioral model	behavioral model
	ontological model	ontological model	ontological model
	interface automata	FSP and LTSs	LTSs with data types
Inference	semantic (ontology)	semantic (ontology)	semantic (ontology)
	ordering constraints	matching sequences	usual ontological relationships
Assumptions	fairness	fairness	no fairness assumption
	synchronous interaction	synchronous interaction	synchronous interaction
	possibility to cache data	possibility to cache data	possibility to cache data
	several actions' mappings	several actions' mappings	several actions' mappings
	use min prefix	use min prefix	use all
	disjoint set I/O actions	no assumption	no assumption
	deterministic automata	deterministic automata	non-deterministic automata
	no hidden actions	no hidden actions	hidden actions
	no info	cycles of length = 1	bounded cycles of length ≥ 1
Outputs	no info	some mismatches	more mismatches
	no info	some compatibility	more inclusive compatibility
	constraints processes	matching processes	LTS with data types

dealt with in [55] are a subset of the ones we cover with our approach since they only consider sequences of "consecutive" actions while we also allow to reason on sequences of "non-consecutive" actions by dealing with the mediation of extra or missing messages. Moreover, to ensure mediator existence, the method in [55] requires that all the protocol interfaces need to be mapped even if they are not used in some compatible behavior that could interoperate under mediation. This suggests that their notion of compatibility is more restrictive than ours. Finally, both works in [54] and [55] produce processes for the mapped I/O actions while we exploit the ontological information and produce an LTS-based specification of the mediator protocol with actual data types, which is amenable to be automatically treated for code generation purposes.

6. Conclusion and future perspectives

Interoperability is a key requirement for heterogeneous protocols within Ubiquitous Computing environments where software systems "*meet dynamically*" and need to *interoperate without a priori knowledge* of each other. Although numerous efforts has been done in many different research areas, protocol interoperability is still an open challenge.

In our research, differently from previous works in the literature, we concentrate on "the elicitation of a way to achieve communication" and, if that exists, on automatically synthesizing protocol mediators enabling correct communication and coordination.

6.1. Final remarks

We proposed a *rigorous method* to automatically reason about software protocols that aim at fulfilling some common goal. The reasoning permits to find a way to achieve communication and correct coordination, and to build the related mediation solution.

We formalized the method for the automated synthesis of protocol mediators by relying on the use of both labeled transition systems and ontologies. Given an arbitrary number of heterogeneous protocols, a mediator enforces their interoperability solving protocol mismatches. The contribution of our work is manyfold: (i) we defined the conditions for mediator existence; (ii) our method automatically synthesizes a correct-by-construction LTS-based specification of the mediator protocol; and (iii) we established the applicability boundaries of our method.

We have started to show, through its application to the real world use case scenario presented in this paper, that our method is viable and sound. The synthesized mediator model is amenable to be automatically treated for code generation purposes.[7]

[7] The code of the mediator for MS and BC, as developed according to our method, is available at: http://www.di.univaq.it/tivoli/sws-challenge-casestudy.zip.

6.2. Limitations of the work and how to cope with them in the future

Dealing with asynchronous protocols. As already mentioned in Section 2, for the purposes of this paper, we modeled synchronous communication only. The application of our approach to systems based on asynchronous communication would require to deal with "hybrid" interaction protocols that can support both synchrony and asynchrony. In previous work from some of the authors [56], we accounted for this notion of hybrid protocol. That is, in order to exchange messages asynchronously, interaction protocols make use of bounded message queues that can be of different sizes for different protocols, in a similar way to what the Abstract Mediator Protocol (see Definition 8) already does. We recall that in order to deal with reordering of messages, the abstract mediator desynchronizes the sending and receiving of messages by using a message buffer. Essentially, achieving applicability to this new setting requires to slightly extend Definition 3 (Behavioral Protocol), Definition 4 (Parallel Composition), and Definition 8 (Abstract Mediator Protocol). In fact, from the one hand, the approach would need to distinguish between synchronous send/receive actions and asynchronous send/receive actions of single protocols. On the other hand, in a composed system, an asynchronous send should model a protocol sending a message to a receiving one; the message would be appended to the message queue of the receiving protocol, unless the queue is full. An asynchronous receive should model a protocol consuming a message from the head of its queue, unless either the queue is empty or the head contains a message different from the one to be consumed. Thus, asynchronous send actions would be blocking only if the queue of the receiving protocol is full. Analogously, asynchronous receive actions would be blocking only if either the protocol queue is empty or a message different from the one to be consumed is on the head of the queue. Synchronous send and receive actions would be kept as they are: the sent message is immediately consumed by the receiving protocol. Thus, no queue would be involved while performing synchronous actions. Finally, the protocols would step over internal actions (i.e., τ) independently. Being local to the single protocol, asynchronous receive actions are not observable in the system and hence they would become τ actions. Thus, the abstract mediator would be equipped with both a h-bounded message buffer for the reordering of messages and a message queue for interacting also with asynchronous protocols via asynchronous send/receive actions. Correctness by construction of the h-bounded mediator would not be affected by these extensions, meaning that it would still hold. In fact, by exploiting theoretical results on realizability [57,58], in the new setting we can still ensure correctness by limiting the size of the mediator's message queue to a bound equal to 1. Note that, instead, the mediator's message buffer can be still kept h-bounded for a generic h. Concerning completeness, it is affected to the same extent it is already limited by reordering. That is, the new synthesis method might not find a mediator that could indeed exist by considering a greater bound for the mediator's message queue.

Applicability of the synthesis method. As discussed in Section 2, our mediator synthesis method assumes to deal with a description of the interaction protocols given in form of automata-based specifications such as LTSs. This assumption is supported by the increasing proliferation of techniques for software model elicitation that generate automata (see [22–27], just to cite a few). In previous work from some of the authors [26], we presented the StrawBerry (Synthesized Tested Refined Automaton of Web-service BEhavior pRotocol) tool, for the automatic discovery of the behavior protocol of a Web Service (WS). Since for a published WS, in practice, only its WSDL description can generally be assumed to be available, StrawBerry derives from the WSDL, in automated way, a partial ordering relation among the invocations of the different WSDL operations, that we represent as an automaton. This automaton, called Behavior Protocol automaton, models the interaction protocol that a client has to abide by to correctly interact with the WS. This automaton explicitly models also the data that has to be passed to the WS operations. More precisely, the states of the behavior protocol automaton are WS execution states and the transitions, labeled with operation names plus I/O data, model possible operation invocations from the client of the WS. The behavior protocol is obtained through synthesis and testing stages. The synthesis stage is driven by data type analysis, through which we obtain a preliminary dependencies automaton, that can be optimized by means of heuristics. Once synthesized, this dependencies automaton is validated through testing against the WS to verify conformance, and finally transformed into the behavior protocol.

StrawBerry is a black-box and extra-procedural method. It is black-box since it takes into account only the WSDL of the WS. It is extra-procedural since it focuses on synthesizing a model of the behavior that is assumed when interacting with the WS from outside, as opposed to intra-procedural methods that synthesize a model of the implementation logic of the single WS operations.

StrawBerry represents one of the possible specification mining tools that can be used in conjunction with our method to achieve better applicability of the mediator synthesis. However, to perform testing with an acceptable accuracy, Strawberry requires the user to build an ad-hoc oracle. When testing cannot be performed, e.g., service providers do not offer a testing version of the services under analysis, the only thing that our method can do is to query the user (e.g., mediator designers, software architects) about confirming or not the semantic correlation of the data mappings inferred by Strawberry during the data type analysis. In this direction, an alternative way that deserves to be investigated in the future regards the ability to automatically check message semantic correlation by exploiting the provided ontological information.

Another aspect that concerns the applicability of our method in the practice of software integration is related to the ability to automatically transform the LTS-based model of the mediator into actual code. There are several ways of doing that since the mediator actual code is unavoidably application specific. In previous work from some of the authors, the mediator model is automatically transformed into different kinds of software artifacts by exploiting model-driven techniques, spanning from COM/DCOM mediator components [36,59] or J2EE ones [60], to modular mediators implemented in Java and

using the Enterprise Integration Patterns (http://www.eaipatterns.com/) provided by the Apache Camel framework (http://camel.apache.org/) [53], or BPEL orchestrators [61,62].

The common idea underlying these different model-to-code transformations consists in navigating the mediator model in order to derive an implementation that reflects the corresponding state machine's structure and behavior. An important aspect, here, is to decouple the mediation logic from the message handling logic. The mediation logic ensures that messages are exchanged according to the interaction protocol of the synthesized mediator model. It mediates the external interaction behavior of the involved services, e.g., for a given service, a certain message must be received before sending another message. The message handling logic is split into message retrieval logic, and message production logic. On the one hand, the retrieval logic allows the mediator to intercept the messages sent by the supervised protocol, hence enabling interoperability with the other protocols. On the other hand, it permits the mediator to manage the storage of previously exchanged messages, whose content might be used by the message production logic to produce the messages to be sent. This allows developers to build mediators in an agnostic way. That is, developers can only focus on coding the message production logic of, e.g., single service operations, by disregarding the message handling, and possible mediation issues and interaction mismatches. All these aspects, i.e., agnostic development of mediators, and automated generation of the needed mediation and message retrieval logic, are supported by our synthesis method that delegates to developers only the implementation of the message production logic within aptly generated skeleton code. That is, the generation of the entire mediation logic is fully automatized. Contrariwise, the generation of the message handling logic cannot be fully automated since it is application specific. However, our approach is able to fully generate the message retrieval logic and a skeleton code for its message production logic. In this way, developers only need to "fill in the blank" of the generated skeletons, completely disregarding interoperability issues.

Concerning the technicalities to transparently interpose the mediator among the protocols to be mediated, they can vary depending on the application domain. For instance, for COM/DCOM mediators, changes in the keys stored into the Windows Registry of the machines hosting the COM/DCOM components to be mediated can be applied [36,59]; for J2EE components, AspectJ can represent a valid alternative as shown in [60]; for mediators in (web-)service-based systems, dynamic binding techniques to change the "invocation address" of a service at deployment- or run-time can be used as discussed in [61,62].

Dealing with data-aware interactions. Our synthesis method automatically generates a model of the mediator that abstracts from the data values exchanged via messages among the mediated protocols. However, in some practical/technological contexts, e.g., (web-)service-based systems, data values can play a crucial role. For instance, the protocol of some services can be implemented by ad-hoc programming, which mixes data used for managing the session with business logic data.[8] By using this technique, web services maintain their state-less nature, and a session (i.e., a conversational protocol with the client) is implicitly realized by passing the relative data (i.e., data encoding the WS state) from one operation to another. Therefore, session data are explicitly added as input/output data of the WSDL operations. By means of this information the session can be kept alive during an entire interaction between the web service and the same client. In order to deal with this technological setting, the mediator can be realized as a BPEL orchestrator, hence exploiting the data correlation constructs directly provided by BPEL. Cleary, this would require to slightly extend the protocol specification that is taken as input by our method with the information about session data for which the generated mediator code has to be able to distinguish different values identifying different clients. Assuming this further piece of information is not a real limitation since, for instance, it is usually provided in the WSDL of a web service by using properties and property alias.

6.3. Further future directions

Beyond the enhancements to the approach that are discussed in Section 6.2 and that we plan to realize in the future, as further future work, we want to devise an alternative mediator synthesis method able to solve communication and coordination mismatches when a domain ontology is not available. In this direction, we are working on the formalization of a method for the automated synthesis of protocol mediators capable of automatically inferring the semantic relations that are now modeled in the domain ontology. In order to do this we require the user to specify the Mazurkiewicz independence relation [63] among common symbols in the protocol signatures. That is, the order relation among symbols joins commutativity. Note that providing the independence relation is a strictly weaker assumption than having a domain ontology in the sense that it is possible to obtain the independence relation from a domain ontology. Hence, when there is a domain ontology the proposed method works but it can work also in cases when the independence relation can be obtained from other information.

In the future, we also plan to (i) study run-time techniques towards efficient synthesis and (ii) ensure dependability. Towards this direction we have already done some investigations by considering both functional interoperability and non-functional interoperability during the synthesis process, i.e., modeling and taking non-functional concerns into account. Preliminary results are presented in [64] with a focus on dependability and performance arising from the execution environment.

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

Acknowledgment

This research work has been supported by: (i) the European Union's H2020 Programme under grant agreement number 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet); (ii) the Ministry of Economy and Finance, Cipe resolution n. 135/2012 (project INCIPICT - INnovating CIty Planning through Information and Communication Technologies); (iii) the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX); (iv) the Software Center through Malmö University, Sweden; and (v) the Knowledge Foundation through the Internet of Things and People research profile at Malmö University, Sweden.

References

[1] M. Weiser, The computer for the 21st century, Sci. Am. 265 (3) (1991) 94–104.
[2] L. Atzori, A. Lera, G. Morabito, The Internet of things: a survey, Comput. Netw. 54 (15) (2010) 2787–2805.
[3] P. Inverardi, M. Autili, D. Di Ruscio, P. Pelliccione, M. Tivoli, Producing software by integration: challenges and research directions (keynote), in: Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 2–12.
[4] K.L. Calvert, S.S. Lam, Formal methods for protocol conversion, IEEE J. Sel. Areas Commun. 8 (1) (1990) 127–142.
[5] S.S. Lam, Correction to "protocol conversion", IEEE Trans. Softw. Eng. 14 (9) (1988) 1376.
[6] K. Okumura, A formal protocol conversion method, in: SIGCOMM, 1986, pp. 30–37.
[7] R. Kumar, S. Nelvagal, S.I. Marcus, A discrete event systems approach for protocol conversion, Discret. Event Dyn. Syst. 7 (3) (1997).
[8] D.M. Yellin, R.E. Strom, Protocol specifications and component adaptors, ACM Trans. Program. Lang. Syst. 19 (1997).
[9] C. Canal, P. Poizat, G. Salaün, Model-based adaptation of behavioral mismatching components, IEEE Trans. Softw. Eng. 34 (4) (2008) 546–563, https://doi.org/10.1109/TSE.2008.31.
[10] B. Benatallah, F. Casati, D. Grigori, H.R.M. Nezhad, F. Toumani, Developing adapters for web services integration, in: Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE), Springer Verlag, Porto, Portugal, 2005, pp. 415–429.
[11] H.R. Motahari Nezhad, G.Y. Xu, 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, ACM, New York, NY, USA, 2010, pp. 731–740.
[12] G. Wiederhold, M. Genesereth, The conceptual basis for mediation services, IEEE Intell. Syst. Appl. 12 (5) (1997) 38–47, https://doi.org/10.1109/64.621227.
[13] R. Vaculín, K. Sycara, Towards automatic mediation of OWL-S process models, in: IEEE International Conference on Web Services, 2007, pp. 1032–1039.
[14] R. Vaculín, R. Neruda, K.P. Sycara, An agent for asymmetric process mediation in open environments, in: R. Kowalczyk, M.N. Huhns, M. Klusch, Z. Maamar, Q.B. Vo (Eds.), SOCASE, in: Lecture Notes in Computer Science, vol. 5006, Springer, 2008, pp. 104–117.
[15] H.R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, F. Casati, Semi-automated adaptation of service interactions, in: WWW '07: Proceedings of the 16th International Conference on World Wide Web, ACM, New York, NY, USA, 2007, pp. 993–1002.
[16] S.K. Williams, S.A. Battle, J.E. Cuadrado, Protocol mediation for adaptation in semantic web services, in: ESWC, 2006, pp. 635–649.
[17] R. Spalazzese, P. Inverardi, V. Issarny, Towards a formalization of mediating connectors for on the fly interoperability, in: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009), 2009, pp. 345–348.
[18] P. Inverardi, V. Issarny, R. Spalazzese, A Theory of Mediators for Eternal Connectors, Proceedings of ISoLA 2010 - 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Part II, vol. 6416, Springer, Heidelberg, 2010, pp. 236–250.
[19] B. Spitznagel, D. Garlan, A compositional formalization of connector wrappers, in: ICSE, 2003, pp. 374–384.
[20] M. Autili, P. Inverardi, F. Mignosi, R. Spalazzese, M. Tivoli, Automated synthesis of application-layer connectors from automata-based specifications, in: A.-H. Dediu, E. Formenti, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications, in: Lecture Notes in Computer Science, vol. 8977, Springer International Publishing, 2015, pp. 3–24.
[21] R.M. Keller, Formal verification of parallel programs, Commun. ACM 19 (7) (1976) 371–384, https://doi.org/10.1145/360248.360251.
[22] D. Lo, L. Mariani, M. Santoro, Learning extended FSA from software: an empirical assessment, J. Syst. Softw. 85 (9) (2012).
[23] D. Lorenzoli, L. Mariani, M. Pezzè, Automatic generation of software behavioral models, in: Proc. of ICSE08, 2008.
[24] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Trans. Software Eng. 27 (2) (2001).
[25] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, A. Zeller, Automatically generating test cases for specification mining, IEEE Trans. Softw. Eng. 38 (2) (2011).
[26] A. Bertolino, P. Inverardi, P. Pelliccione, M. Tivoli, Automatic synthesis of behavior protocols for composable web-services, in: Proc. of ESEC/FSE, 2009.
[27] H. Raffelt, B. Steffen, T. Berg, T. Margaria, Learnlib: a framework for extrapolating behavioral models, Int. J. Softw. Tools Technol. Transf. 11 (5) (2009).
[28] R. Milner, Communication and Concurrency, PHI Series in Computer Science, Prentice Hall, 1989.
[29] S. Uchitel, J. Kramer, J. Magee, Incremental elaboration of scenario-based specifications and behavior models using implied scenarios, ACM Trans. Softw. Eng. Methodol. 13 (1) (2004) 37–85.
[30] T. Margaria, The semantic web services challenge: tackling complexity at the orchestration level, in: ICECCS'08, 2008.
[31] H. Lausen, U. Küster, C. Petrie, M. Zaremba, S. Komazec, SWS Challenge Scenarios, Springer US, Boston, MA, 2009, pp. 13–27.
[32] R. Spalazzese, P. Inverardi, Mediating connector patterns for components interoperability, in: ECSA, 2010, pp. 335–343.
[33] U. Aßmann, S. Zschaler, G. Wagner, Ontologies, Meta-Models, and the Model-Driven Paradigm, Springer, 2006.
[34] Y. Kalfoglou, M. Schorlemmer, Ontology mapping: the state of the art, Knowl. Eng. Rev. 18 (1) (2003).
[35] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, The Description Logic Handbook, Cambridge University Press, 2003.
[36] M. Tivoli, P. Inverardi, Failure-free coordinators synthesis for component-based architectures, Sci. Comput. Program. 71 (3) (2008) 181–212.
[37] M. Autili, C. Chilton, P. Inverardi, M.Z. Kwiatkowska, M. Tivoli, Towards a connector algebra, in: Leveraging Applications of Formal Methods, Verification, and Validation, ISoLA 2010, Heraklion, Crete, Greece, 18–21 October, 2010, Proceedings, Part II, 2010, pp. 278–292.
[38] R. Davis, H. Shrobe, P. Szolovits, What is a knowledge representation?, AI Mag. 14 (1) (1993) 17–33.
[39] J.F. Sowa, Knowledge Representation: Logical, Philosophical and Computational Foundations, Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2000.
[40] R. Brachman, H. Levesque, Knowledge Representation and Reasoning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
[41] F. van Harmelen, F. van Harmelen, V. Lifschitz, B. Porter, Handbook of Knowledge Representation, Elsevier Science, San Diego, USA, 2007.
[42] S. Staab, R. Studer (Eds.), Handbook on Ontologies, International Handbooks on Information Systems, Springer, 2004.
[43] R. Studer, V.R. Benjamins, D. Fensel, Knowledge engineering: principles and methods, Data Knowl. Eng. 25 (1–2) (1998) 161–197, https://doi.org/10.1016/S0169-023X(97)00056-6.
[44] F. Baader, I. Horrocks, U. Sattler, Description Logics, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 21–43.
[45] Y. Kalfoglou, M. Schorlemmer, Ontology mapping: the state of the art, Knowl. Eng. Rev. 18 (1) (2003) 1–31, https://doi.org/10.1017/S0269888903000651.

[46] S. Pavel, J. Euzenat, Ontology matching: state of the art and future challenges, IEEE Trans. Knowl. Data Eng. 25 (1) (2013) 158–176, https://doi.org/10.1109/TKDE.2011.253.
[47] G. Wiederhold, Mediators in the architecture of future information systems, IEEE Computer 25 (1992) 38–49.
[48] L. Cavallaro, E.D. Nitto, M. Pradella, An automatic approach to enable replacement of conversational services, in: ICSOC/ServiceWave, 2009.
[49] M. Stollberg, E. Cimpian, A. Mocan, D. Fensel, A semantic web mediation architecture, in: Proceedings of the 1st Canadian Semantic Web Working Symposium, CSWWS 2006, Springer, 2006.
[50] E. Cimpian, A. Mocan, Wsmx process mediation based on choreographies, in: C. Bussler, A. Haller (Eds.), Business Process Management Workshops, vol. 3812, 2005, pp. 130–143.
[51] R. Passerone, L. de Alfaro, T.A. Henzinger, 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, 2002, pp. 132–139.
[52] R. Bruni, I. Lanese, U. Montanari, A basic algebra of stateless connectors, Theor. Comput. Sci. 366 (1) (2006) 98–120, https://doi.org/10.1016/j.tcs.2006.07.005.
[53] P. Inverardi, M. Tivoli, Automatic synthesis of modular connectors via composition of protocol mediation patterns, in: Proceedings of ICSE'13, 2013.
[54] A. Bennaceur, C. Chilton, M. Isberner, B. Jonsson, Automated mediator synthesis: combining behavioural and ontological reasoning, in: Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, 25–27 September, 2013, Proceedings, 2013, pp. 274–288.
[55] A. Bennaceur, V. Issarny, Automated synthesis of mediators to support component interoperability, IEEE Trans. Softw. Eng. 41 (3) (2015) 221–240, https://doi.org/10.1109/TSE.2014.2364844.
[56] M. Autili, P. Inverardi, M. Tivoli, Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates, Sci. Comput. Program. 160 (2018) 3–29, https://doi.org/10.1016/j.scico.2017.10.010.
[57] S. Basu, T. Bultan, M. Ouederni, Deciding choreography realizability, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, 2012, pp. 191–202.
[58] S. Basu, T. Bultan, Automated choreography repair, in: Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 13–30.
[59] M. Tivoli, M. Autili, Synthesis, a tool for synthesizing correct and protocol-enhanced adaptors, RSTI - L'objet, Coordination and Adaptation Techniques 12 (1) (2006) 77–103.
[60] M. Autili, L. Mostarda, A. Navarra, M. Tivoli, Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems, J. Syst. Softw. 81 (12) (2008) 2210–2236.
[61] M. Autili, P. Inverardi, M. Tivoli, Automated synthesis of service choreographies, IEEE Softw. 32 (1) (2015) 50–57, https://doi.org/10.1109/MS.2014.131.
[62] M. Autili, A. Di Salle, F. Gallo, C. Pompilio, M. Tivoli, Aiding the realization of service-oriented distributed systems, in: 34th Annual ACM Symp. on Applied Computing, SAC '19, 2019.
[63] V. Diekert, G. Rozenberg, The Book of Traces, World Scientific, 1995.
[64] N. Nostro, R. Spalazzese, F.D. Giandomenico, P. Inverardi, Achieving functional and non functional interoperability through synthesized connectors, J. Syst. Softw. 111 (2016) 185–199.