Components Interoperability through Mediating Connector Patterns

Spalazzese Romina, Paola Inverardi

Components Interoperability through Mediating Connector Patterns*†

Romina Spalazzese Paola Inverardi
Università degli Studi dell'Aquila
via Vetoio I-67100 L'Aquila, Italy
romina.spalazzese@di.univaq.it paola.inverardi@di.univaq.it

A key objective for ubiquitous environments is to enable system interoperability between system's components that are highly heterogeneous. In particular, the challenge is to embed in the system architecture the necessary support to cope with behavioral diversity in order to allow components to coordinate and communicate. The continuously evolving environment further asks for an automated and on-the-fly approach. In this paper we present the design building blocks for the dynamic and on-the-fly interoperability between heterogeneous components. Specifically, we describe an Architectural Pattern called *Mediating Connector*, that is the key enabler for communication. In addition, we present a set of *Basic Mediator* Patterns, that describe the basic mismatches which can occur when components try to interact, and their corresponding solutions.

1 Introduction

A multitude of heterogeneous networked devices are today embedded in the Ubiquitous networked environment [6] where a key objective is to enable system interoperability between system's components that are highly heterogeneous.

Tremendous work has been done in the middleware field to enable diverse networked systems to actually work together, especially to respond to the new needs introduced by the ubiquitous environments. Devices need to automatically detect services available in the environment and adapt their own communication protocols to interoperate with them since networked applications are developed on top of diverse middleware. However, the proposed solutions only address middleware-layer interoperability letting still open the interoperability at application-layer that calls for *mediating connectors* or *mediators* for short.

The mediator concept was initially introduced to cope with the integration of heterogeneous data sources [21, 22], and as design pattern [26]. Moreover, mediators have received an increasing attention within the Web Services and Semantic Web contexts. With the significant development of Web technologies, a big effort has been devoted to mediate many heterogeneity dimensions, spanning [16]: terminology, representation format and transfer protocols, functionality and application-layer protocols. Protocol mediation is further concerned with behavioral mismatches that may occur during interactions. Other approaches that share the same formal settings as protocol mediation have been proposed to solve mismatches in the field of supervisory control theory of discrete event systems [14]. More recently, in the field of software architectures ad hoc wrappers have been proposed to address communication problems as for example data translation, connectors combination, role addition, extra functional deficiencies [15]. A lot of work has been also devoted to behavioral adaptation, [3, 25, 29] to mention few. Automated mediation has deserved attention, especially in the context of Web services and Semantic Web technologies

*This work is an extension of our previous work [8].

†This work is partly supported by the CONNECT European Project No 231167.

Cámara, Canal, and Salaün (Eds.)
Component and Service Interoperability (WCSI10)
EPTCS 37, 2010, pp. 27–41, doi:10.4204/EPTCS.37.3

[12, 13, 7, 9]. More recently the challenge is to provide general solutions to the behavioral diversities at runtime and on-the-fly, to respond to the continuous evolution of the environment [10].

All these problems call for an approach to protocol mediation based on the categorization of the types of protocol mismatches that may occur and that must be solved in order to provide corresponding solutions to these recurring problems. This immediately reminds of patterns [1, 4, 20, 26]. In this paper we present a set of design building blocks for the interoperability between heterogeneous components. A catalogue of problems and their related solutions would not solve all the possible mismatches but it would certainly facilitate the solution.

The contributions of this paper are: (1) an Architectural Pattern called *Mediating Connector*, that is the key enabler for communication; (2) a set of *Basic Mediator* Patterns that describe: (i) the basic mismatches which can occur while components try to interact, and (ii) their corresponding solutions. This work extends a preliminary version [8] with: the introduction of a related work section, the introduction of a detailed description of basic mismatch patterns and a extended description of the example including a new section on the patterns application on it. This work is part of the CONNECT European Project [10] where we investigate an overall framework for the seamless networking of heterogeneous systems.

The remaining of the paper is organized as follows. In Section 2, we introduce a motivating example for protocol mediation. In Section 3, we describe a pattern-based approach which we envision for the automatic synthesis of mediating connectors for the ubiquitous networked environment. We illustrate the Mediating Connector Architectural Pattern in Section 4. In Section 5, we illustrate the Basic Mediator Patterns including the basic interoperability mismatches, which can occur when two heterogeneous components try to interoperate, and their respective solutions. In Section 6, we show the application of the mediator patterns to the example to solve the interoperability mismatches. Then, we discuss related work in Section 7 and we conclude, in Section 8, by also outlining future work.

2 Example Motivating the Mediation need

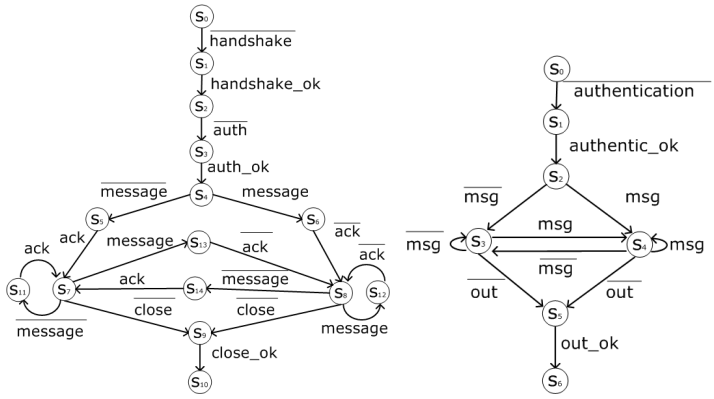
To better illustrate *protocol mediation*, and to make the underlying problem more concrete, in the following we describe the example used in [18] where we have been studying the problem and first results on the theory our approach is based on appeared. An extended and more complete version of the theory can be found [17]. We consider the simple yet challenging example of instant messaging. Various instant messaging systems are now in use, facilitating communications among people. However, although those systems implement similar functionalities, end-users need to use the very same system to communicate due to behavioral mismatches of the respective protocols.

For instance, consider Windows Messenger[1](WM), and Jabber Messenger[2](JM). Figure 1 models the respective behaviors of the associated protocols using Labeled Transition Systems (LTS) [23]. LTSs constitute a widely used model for concurrent computation and are often used as a semantic model for formal behavioral languages such as process algebras. We use the usual convention that actions with overbar denote output actions while the ones with no overbar denote input actions. These systems should be able to interoperate since they both amount to supporting authentication with their servers and then message exchanges among peers. However mediating their respective protocols to achieve interoperability is far from trivial, especially if one wants to achieve a general solution.

An effort has been done in [19] to mediate instant messaging protocol mismatches allowing communication between any two clients. Unfortunately, the proposed solution requires the implementation of

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

[2]Jabber Software Foundations, http://www.jabber.org/



(a) Windows Messenger protocol (b) Jabber protocol

Figure 1: Behavioral modeling of two instant messaging protocols

the translation from any client protocol (to be supported) to a reference exchange protocol to be given, and vice versa. This obviously affects the generality and the automation of the approach. Several works have devoted attention to protocol mediation in the context of Web services using patterns [5],[2],[30]. In particular the latter provides tools to developers to help them to identify protocol mismatches and to compose mediators. However, this remains quite limited with respect to enabling interoperability in today's networking environments that are highly dynamic. Indeed, mediators need to be synthesized on demand so as to allow interactions with networked systems that are not known in advance. Such a concern is in particular recognized by the Web service research community that has been studying solutions to the automatic mediation of business processes in the recent years.

3 A Pattern-based Approach for Interoperability Mismatches

In this section we describe our proposal for an automated pattern based approach. It is based on first results describing a theory of mediators at a high level [18]. For the sake of this paper, we make some assumptions and we investigate the related underling research problems as part of CONNECT. We assume to know the interaction protocols run by two networked components as Labeled Transition Systems (LTS) and the components' interfaces with which to interact as advertised or as result of learning techniques [11, 28]. We also assume a semantic correspondence between the messages exchanged among components exploiting ontologies. The first step is to establish whether the components are *potentially compatible*, that is, if it makes sense for them to *interoperate* through the Mediating Connector. This amounts to understand if the components share *intents* (traces), i.e., if, modulo some adaptation, they show complementary sequences of messages visible at interface level. To do this we envision (1) a decomposition strategy/tool to decompose the whole components' behavior (LTS) into *elementary behaviors* (traces) representing *elementary intents* of the components and (2) an automatic analyzer to identify mismatches between elementary behaviors of the different components as done in other research areas [3, 24]. Once discovered the components compatibility, solving their interoperability means solving the behavioral mismatches that they exhibit. Then it is necessary to: (3) define a mismatches manager to solve the identified mismatches between elementary behaviors; (4) define a composition approach to build elementary mediating behaviors (mediating traces) based on the identified mismatches and their relative solutions; (5) define a composition strategy to build a mediating connector's behavior starting

from the elementary mediating behaviors.

The above described approach is far from trivial, especially to achieve automatically. However, in the following we show its feasibility. To address steps (1) and (5) the approach makes use of a compositional strategy to decompose components interaction protocols into traces and compose mediating connectors interaction protocol from mediating traces respectively. Furthermore, we describe six Basic Mediator Patterns that are the building blocks on which the steps (2), (3), and (4) can be built upon.

4 Mediating Connector Architectural Pattern

The interoperability problem between diverse components populating the ubiquitous environment and its related solution is characterized as a *Mediating Connector Architectural Pattern* basing on the template used in [4] that contains the following fields: Name, Also Known As, Example, Context, Problem, Solution, Structure, Dynamics, Implementation, Example Resolved, Variants, Consequences. The Mediating Connector is a behavioral pattern and represents the architectural building block embedding the necessary support to dynamically cope with components' behavioral diversity.

Name. Mediating Connector.

Also Known As. Mediator.

Example. Consider the ubiquitous environment that embeds networked devices from a multitude of applications domain, for example consumer electronics or mobile and personal computing devices. Suppose that *potentially compatible* applications running on various devices want to *interoperate*. Potentially compatible applications are applications that may share some *intent* resulting in complementary portions of their *interaction protocols*, i.e., complementary sequences of messages visible at interface level. In principle those applications should be able to interoperate, but because of some *behavioral differences* that they exhibit, they are not compatible. With interoperate we mean *coordinate* and *communicate (i.e. synchronize)*. For example, consider the instant messaging applications described in Section 2. Users of different messengers may want to communicate and in principle this should be possible since the two different messengers implement similar functionalities. However they do it in different ways and this prevents the communication.

Context. The environment is distributed and changes continuously. Heterogeneous (mismatching) systems populating the environment require seamless coordination and communication.

Problem. In order to support existing and future systems' interoperability, some means of mediation is required. From the components' perspective, there should be no difference whether interacting with a peer component, i.e, using the very same interaction protocol, or interacting through a mediator with another component that uses a different interaction protocol. The component should not need to know anything about the protocol of the other one while continuing to "speak" its own protocol.

Using the Mediating Connector, the following *forces* (aspects of the problem that should be considered when solving it [4]) need to be balanced: (a) the different components should continue to use their own interaction protocols. That is components should interact as if the Mediating Connector were transparent; (b) the following *basic interaction protocol mismatches* should be solved in order for a set of components to coordinate and communicate (a detailed description of these mismatches is given within Section 5): 1) Extra Send/Missing Receive Mismatch; 2) Missing Send/Extra Receive Mismatch; 3) Signature Mismatch; 4) Ordering Mismatch; 5) One Send-Many Receive/Many Receive-One Send Mismatch; 6) Many Send-One Receive/One Receive-Many Send Mismatch

Solution. The introduction of a Mediating Connector to manage the interaction behavioral differences between potentially compatible components. The idea behind this pattern is that, by using the Mediating

Connector, components that would need some interaction protocol's adaptation to become compatible, and hence to interoperate, are able to coordinate and communicate achieving their goals/intents without undergoing any modification.

The Mediating Connector is one (or a set of) component(s) that manage the behavioral mismatches listed above. It directly communicates with each component by using the component's proper protocol. The mediator forwards the interaction messages from one component to the other by making opportune translation/adaptation of protocols when/if needed.

Structure. The Mediating Connector Pattern comprises three types of participating components: communicating components, potentially compatible components and mediators.

The *communicating components* implement already compatible components, i.e. components able to interact and evolve following their usual interaction behavior. The *potentially compatible components* implement the application level entities (whose behavior, interfaces' description and semantic correspondences are known). Each component wants to reach its intents by interacting with other components able to satisfy its needs, i.e. required/provided functionalities. However the components are unable to directly interact because of protocol mismatches. Thus, the potentially compatible components can only evolve following their usual interaction behavior, without any change. The *mediators* are entities responsible for the mediated communication between the components. This means that the role of the mediator is to make compatible components that are mismatching. That is, a mediator must receive and properly forward requests and responses between potentially compatible components that want to interoperate. Figure 2 shows the object involved in a mediated system.

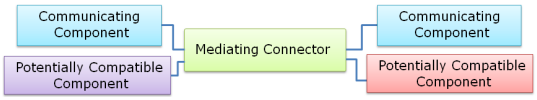


Figure 2: Entities involved in a mediated system

Dynamics. Figure 3 illustrates the interactions between three components and one mediator belonging to the messengers example. Triggered by a user, the WM protocol (Figure 1(a)) performs one of its possible behavior: it authenticates after an handshake, sends/receives several messages, and closes. The mediator should: (1) forward the handshake and authentication messages as they are between the WM and its authentication server (communicating components), (2) translate and forward messages between the WM and JM (potentially compatible components), and (3) forward the closing messages as they are between the WM and its server (communicating components). With the term "translation" we mean not just a language translation but also a "behavioral translation" (see Section 5 for details).

Implementation. The implementation of this pattern implies the definition of an approach/tool (we have proposed one in Section 3) to automatically synthesize the behavior of the Mediating Connector which allows the potentially compatible components to interoperate mediating their interactions.

Example Resolved. The Mediating Connector's concrete protocol for the example is shown in Figure 10. Once established that they are potentially compatible (i.e. they have some complementary portion of interaction protocols), the mediating connector manages the components' behavioral mismatches allowing them to have a mediated coordination and communication.

Variants. Distributed Mediating Connector. It is possible to implement this pattern either as a centralized component or as distributed components, that is by a number of smaller components. This introduces a synchronization issue that has to be taken into consideration while building the mediator behavior.

Consequences. The main *benefit* of the Mediating Connector Pattern is that it allows interoperability between components that otherwise would not be able to do it because of their behavioral differences.

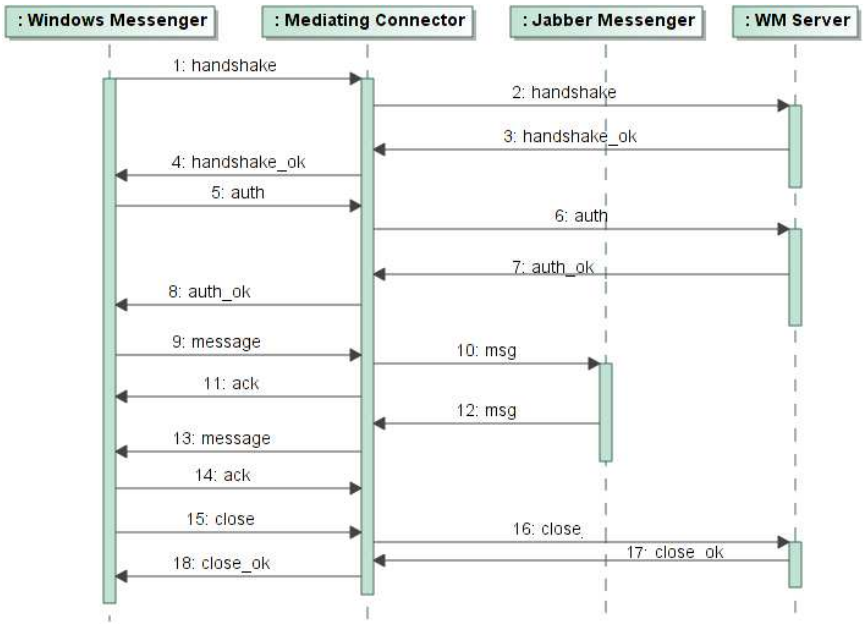


Figure 3: Scenario on the relevant operation of a Mediator

These components do not use the very same observable protocols and this prevents their cooperation while, implementing similar functionalities, they should be able to interact. The main *liability* that the Mediating Connector Pattern imposes is that systems using it are slower than the ones able to directly interact because of the indirection layer that the Mediating Connector Pattern introduces. However the severity of this drawback is mitigate and made acceptable by the fact that such systems, without mediator, are not able at all to interoperate.

5 Basic Mediator Patterns

In the previous sections we characterized the Mediating Connector pattern and we sketched an approach for the automatic synthesis of its behavior.

In this section, we concentrate on six finer grain *Basic Mediator Patterns* which represent a systematic approach to solve interoperability mismatches that can occur during components' interaction. The Basic Mediator Patterns are constituted by basic interoperability mismatches with their corresponding solutions and are: (1) Message Consumer Pattern, (2) Message Producer Pattern, (3) Message Translator Pattern, (4) Messages Ordering Pattern, (5) Message Splitting Pattern, (6) Messages Merger Pattern.

The mismatches, inspired by service composition mismatches, represent send/receive problems that can occur while synchronizing two traces. In this paper we are not considering parameters mismatches which are extensively addressed elsewhere [27]. Figure 4 shows the basic interoperability mismatches that we explain in detail in the following. For each basic interoperability mismatch, we consider two traces (left and right) coming from two potentially compatible components. All the considered traces are the most elementary with respect to the messages exchanged and only visible messages are shown.

It is obvious that, in real cases, the traces may also contain portions of behavior already compatible (abstracted by dots in the figure) and may amount to any combination of the presented mismatches.

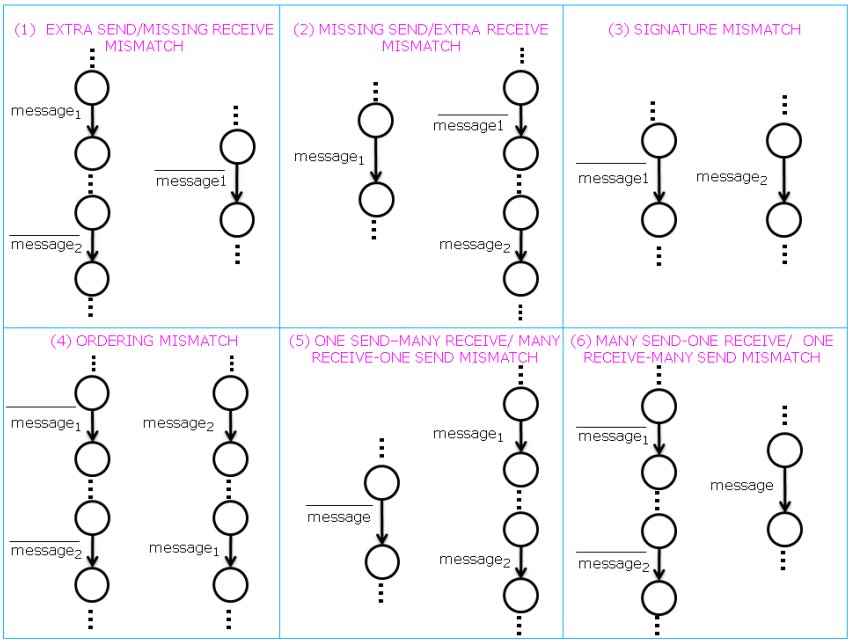


Figure 4: Basic interoperability mismatches

Then an appropriate strategy to detect and manage this is needed. The considered basic mismatches are addressed by the basic solutions (elementary mediating behaviors) illustrated in Figure 5 where only their visible messages are shown (messages that they exchange with the components).

The six Basic Mediator Patterns share the context, i.e., the situation in which they may apply and have a unique intent.

Context. Considering two traces (left and right) expressing similar complementary functionalities. Focus on one of their semantically equivalent elementary actions.

Intent. To allow synchronization between the two traces letting them evolve together which otherwise would not be possible because of behavioral mismatches.

(1) MESSAGE CONSUMER PATTERN.

Problem. (1) Extra send/missing receive mismatch ((1) in Figure 4, where the extra send action is $\overline{message_2}$). One of the two considered traces either contains an extra send action or a receive action is missing.

Example. Consider two traces implementing the abstract action "send (respectively receive) message". For example, in the mismatch (1) of Figure 4 the right trace implements only the sending of the message while the left trace implements the receiving of the message and the sending of an acknowledgment ($\overline{message_2}$).

Solution. Introducing a *message consumer* (solution of mismatch (1) in Figure 5) that is made by an action that, "consumes" the identified extra send action by synchronizing with it, letting the two traces communicate.

Example Resolved. First the two traces synchronize on the sending/receiving of the message ($message_1$) and then the left trace synchronizes its sending of the acknowledgment ($\overline{message_2}$) with the message consumer that receives it.

Variants. Possible variants and respective solutions are represented in Figure 6 and are:

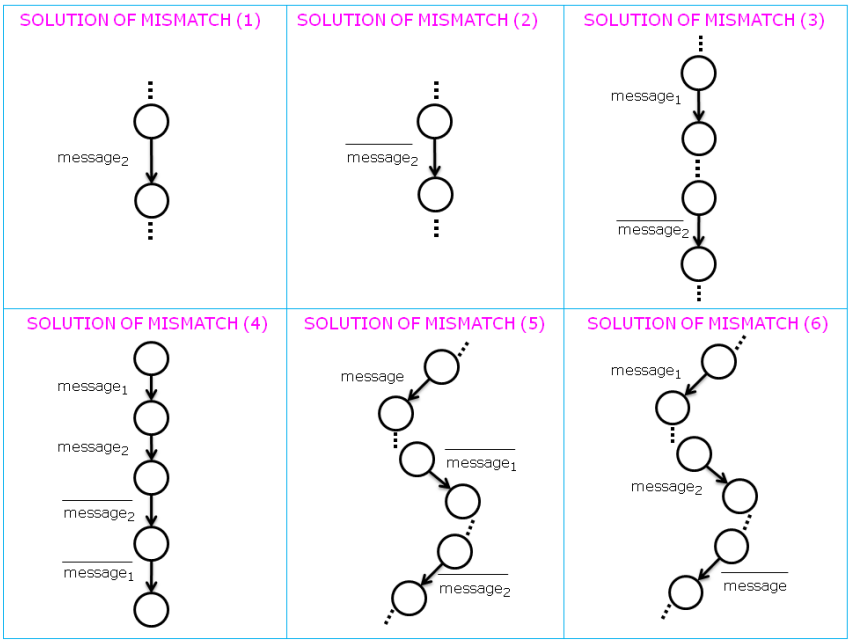


Figure 5: Basic solutions for the basic mismatches

(a) $message_1$ has exchanged send/receive type within the two traces, i.e., the left trace is the sequence $\overline{message_1}.\overline{message_2}$ while the right trace is just $message_1$. In this case the message consumer remains the same ($message_2$).

(b) $\overline{message_1}$ is the extra send message instead of $\overline{message_2}$. The left trace is the sequence $\overline{message_1}.\overline{message_2}$ while the right trace is made by $message_2$. In this case the message consumer performs $message_1$.

(c) the extra send message is $\overline{message_1}$, the left trace is the sequence $\overline{message_1}$. $message_2$ while the right trace is $\overline{message_2}$. In this case the message consumer is made by $message_1$.

(2) MESSAGE PRODUCER PATTERN.

Problem. (2) Missing send/extra receive mismatch ((2) in Figure 4, where the missing send action is $\overline{message_2}$). One of the two considered traces either contains an extra receive action or a send action is missing in it. This is the dual problem of mismatch (1).

Example. Consider two traces implementing the abstract action "send (respectively receive) message". In the mismatch (2) of Figure 4, the right trace implements the sending of the message ($\overline{message_1}$) and the receiving of an acknowledgment ($message_2$) while the left trace implements just the receiving the message ($message_1$).

Solution. Introducing a *message consumer* (solution of mismatch (2) in Figure 5) made by an action that "produces" the missing send action corresponding to the identified extra receive action and let the two traces synchronize.

Example Resolved. The two traces first synchronize on the sending/receiving of the message ($message_1$) and then the right trace synchronize its receive of the acknowledgment ($message_2$) with the message consumer mediator that sends it.

Variants. Possible variants and respective solutions are shown in Figure 7 and are:

(a) $message_1$ has exchanged send/receive type within the two traces, i.e., the left trace is $\overline{message_1}$

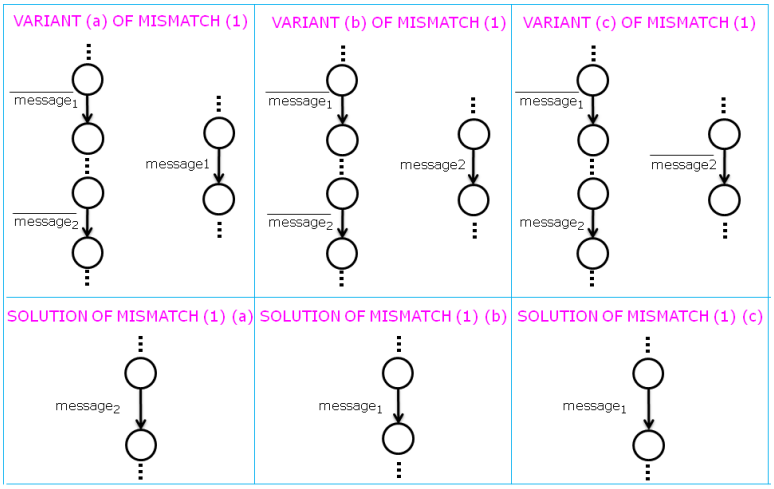


Figure 6: Variants of the Basic Mediator Pattern (1)

while the right trace is the sequence $message_1.message_2$ In this case the message producer performs $\overline{message_2}$.

(b) the missing send message is $\overline{message_1}$, instead of being $message_2$, the right trace is the sequence $message_1.\overline{message_2}$ while the left trace is made by $message_2$. In this case the message producer is made by $\overline{message_1}$.

(c) the missing send message is $message_1$, the left trace is $message_2$ while the right trace is the sequence $message_1.\overline{message_2}$. In this case the message producer is made by $\overline{message_1}$.

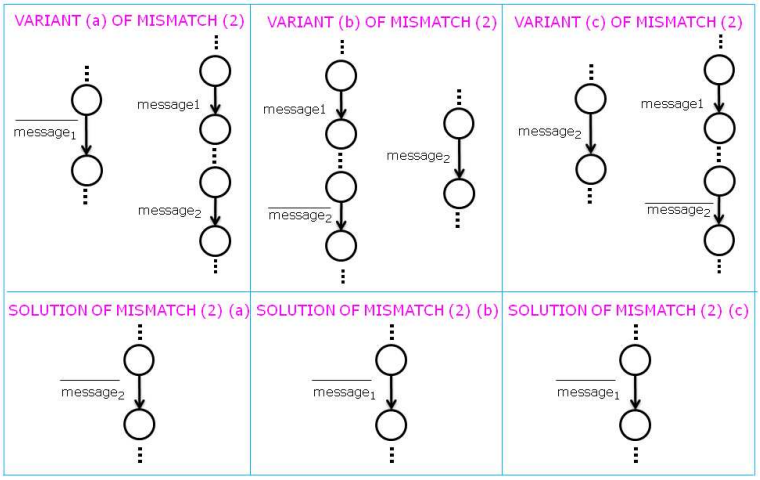


Figure 7: Variants of the Basic Mediator Pattern (2)

(3) MESSAGE TRANSLATOR PATTERN.
Problem. (3) Signature mismatch (upper right box of Figure 4). The two traces represent semantically complementary actions but with different signatures. With signature we mean only the action name.

Example. Consider two traces implementing the abstract action "send (respectively receive) information request". Instantiating the mismatch (3) of Figure 4, $\overline{message_1}$ could be the send of a message *Information* while $message_2$ the receive of a *Request* message.
Solution. Introducing a *message translator* (solution of mismatch (3) in Figure 5). It receives the request and sends it after a proper translation. We assume the existence of some entity able to do the translation[3]. Referring to the example, the translator mediator trace is: $message_1.\overline{message_2}$.
Example Resolved. First the message *Information* is exchanged between one trace and the mediator. After its translation, a *Request* message is sent by the mediator to the other trace. The message translator performs: $Information.\overline{Request}$.
Variants. A possible variant with its solution is shown in Figure 8 and amount to exchange sender/receiver roles between the two traces, i.e., $message_1$ and $\overline{message_2}$ and the solution is made by $message_2.\overline{message_1}$.
(4) MESSAGES ORDERING PATTERN.

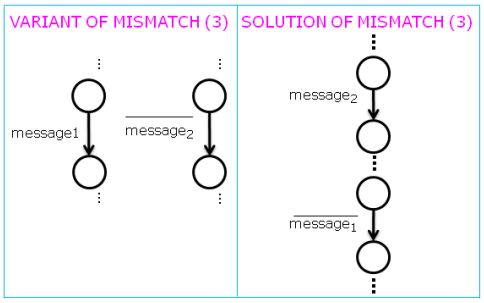


Figure 8: Variants of the Basic Mediator Pattern (3)

Problem. (4) Ordering mismatch ((4) in Figure 4, where both traces perform complementary (send/receive) $message_1$ and $message_2$ but in different order). Both traces consist of complementary functionalities but they perform the actions in different orders. Nevertheless this mismatch can be considered also as a combination of extra/missing send/receive actions mismatches (1) and (2), however we choose to consider it as a first class mismatch. Generally speaking, it may happen that not all the ordering problems are solvable due to the infinite length of the traces. However this is not our case.
Example. Consider two traces implementing the abstract action "send (respectively receive) name". $message_1$ and $message_2$ in the mismatch (4) of Figure 4, for example, correspond to $FirstName$ and $LastName$ respectively. Then, one sends the sequence $FirstName.LastName$ while the other receives $LastName.FirstName$.
Solution. Introducing a *messages ordering* (solution of mismatch (4) in Figure 5). This pattern has a compatible behavior for both the traces. The pattern is made by a trace that receives the messages and, after a proper reordering, resends them.
Example Resolved. Referring to the example, the messages ordering trace is: $message_1.message_2$. $\overline{message_2}$. $\overline{message_1}$ that is $FirstName.LastName.\overline{Last}$- $\overline{Name}.\overline{FirstName}$. That is, first one trace synchronizes with the mediator which receives the messages and then the mediator reorders the messages and sends them to the other trace.
Variants. Possible variantsand respective solutions are shown in Figure 9 and are:

(a) left trace has exchanged sender/receiver role with respect to the right trace, i.e., the left trace is

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

the sequence $\overline{message_2}.\overline{message_1}$ while the right trace is the sequence $message_1.message_2$. In this case the messages ordering is the sequence $message_2.message_1.\overline{message_1}.\overline{message_2}$.

(b) in both traces the first action is a send while the second is a receive. That is, the left trace is $\overline{message1}.message2$ while the right is $\overline{message2}.message1$. In this case the message ordering is the sequence $message_1.message_2$. $\overline{message_2}.\overline{message_1}$.

(c) in both traces the first action is the receive followed by the send. That is, the left trace is $message_1$. $\overline{message_2}$ while the right is $message_2.\overline{message_1}$. In this case the basic solution to solve the mismatch is not the messages ordering. It is a proper combination of messages producers and consumers (message producer followed by message consumer for the left trace followed by message producer followed by message consumer for the right trace). That is, $\overline{message_1}.message_2$ followed by $\overline{message_2}.message_1$.

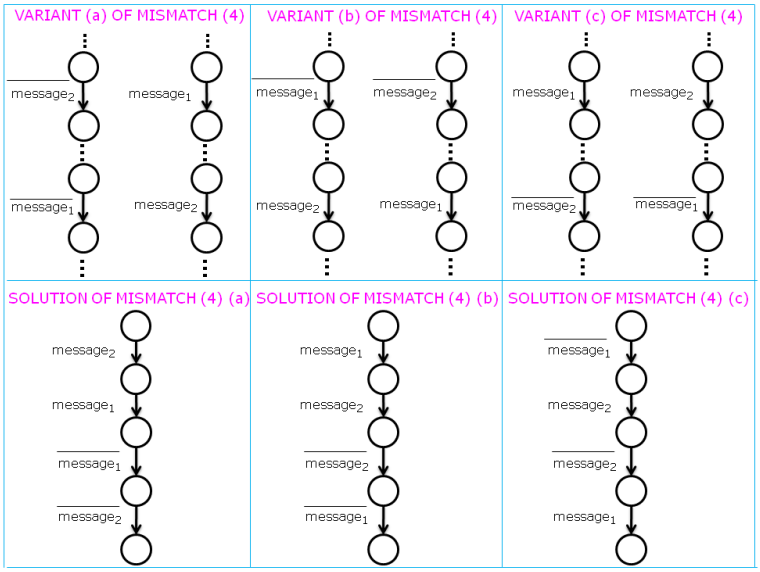


Figure 9: Variants of the Basic Mediator Pattern (4)

(5) MESSAGE SPLITTING PATTERN.

Problem. (5) One send-many receive/many receive-one send mismatch ((5) in Figure 4). The two considered traces represent a semantically complementary functionality but one expresses it with one action and the other with two actions.

Example. Consider two traces implementing the abstract action "send (respectively receive) name". Instantiating the one send-many receive mismatch (5) of Figure 4, for example, $\overline{message}$ can be the send of one message $\overline{FirstLastName}$ while $message_1$ and $message_2$ are the receive of two separate messages $FirstName$ and $LastName$.

Solution. Introducing a *messages splitting* (solution of mismatch (5) in Figure 5). It receives one message from one side, splits it properly, and sends the split messages to the other. We assume the existence of some entity able to do the splitting operation[4]. Referring to the example, the trace of the message

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

splitting (5) is: $message.\overline{message_1}.\overline{message_2}$.

Example Resolved. With respect to the example, the mediator first performs one receive, then a splitting, and subsequently sends two messages. That is, of $FirstLastName.\overline{FirstName}.\overline{LastName}$.

(6) MESSAGE MERGER PATTERN.

Problem. (6) Many send-one receive/one receive-many send mismatch ((6) in Figure 4). The two considered traces represent a semantically complementary functionality but they express it with a different number of actions. This is the dual problem of mismatch (5).

Example. Consider two traces implementing the abstract action "send (respectively receive) name". Instantiating the many send-one receive (6) of Figure 4, for example, $\overline{message_1}$ and $\overline{message_2}$ are the sending of two separate messages $\overline{FirstName}$ and $\overline{LastName}$ while *message* is the receiving of $FirstLastName$.

Solution. Introducing a *messages merging* (solution of mismatch (6) in Figure 5). It receives two messages from one side, merges them properly, and sends the merged messages to the other. We assume the existence of some entity able to do the merge operation. Referring to the example, the trace of the messages merging is: $message_1.message_2.\overline{message}$.

Example Resolved. With respect to the example, the mediator first performs two receives, then a merge, and subsequently sends one message. That is, $FirstName.LastName.\overline{FirstLastName}$.

6 Application of the Patterns to the example

The aim of this section is to show the patterns at work, putting together all the jigsaws puzzle. Thanks to the compatibility analyzer, we discover that the two messengers components are potentially compatible, i.e., our instant messengers share some intent having complementary portion of interaction protocols. Hence, it makes sense to use the architectural Mediating Connector Pattern to mediate their conversations. Following the pattern-based approach described in Section 3, the messengers behavior is decomposed into traces representing elementary behaviors. Then, the traces are analyzed and their basic mismatches are identified thanks to the basic mediators patterns. Subsequently, a composition strategy is applied to build elementary mediators, i.e., mediator traces, exploiting the basic mediators patterns. Finally, in order to automatically synthesize the behavior of the whole Mediating Connector for the messengers, a composition approach aggregates the elementary mediators so to have a mediated coordination and communication.

Figure 10 shows the behavior of the Mediating Connector for the messengers. The mediator, in this

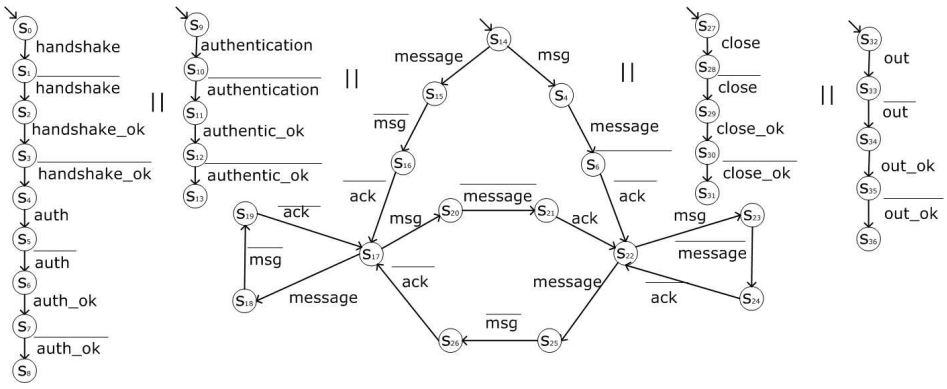


Figure 10: Behavioral description of the Mediating Connector for the messengers example

example, allows the interaction between the two different messengers by translating conversations from one protocol into the other and forwarding them while only forwarding the interactions between each messenger and its server. In particular, the Windows Messenger uses acknowledged messages while the Jabber Messenger does not use acknowledges. This is detected and solved by the mediator thanks to the application of the Message Consumer Pattern and Message Producer Pattern, from which messages "$\overline{ack}$" and "*ack*" originate respectively.

7 Related work

In the last decades the notion of mediators has been investigated in several contexts. Initially introduced to cope with the integration of heterogeneous data sources [21],[22] and as design pattern [26], it has been subsequently studied within the Web context to cope with many heterogeneity dimensions [16] and with behavioral mismatches that may occur during interactions between business processes (protocol mediation). An approach to protocol mediation is to categorize the types of protocol mismatches that may occur and that must be solved in order to provide corresponding solutions to these recurring problems. This immediately reminds of patterns whose pioneer was Alexander [1]. Patterns have received attention in several research areas: for example Bushmann et al. [4] have defined patterns for software architectures; more recently in [20] a pattern language has been proposed revisiting existing architectural patterns; and the "gang of four" in [26] have defined design patterns. Between all, two design patterns are related to our: the Mediator Design Pattern, which is a behavioral pattern, and the Adapter which is structural. The first one is similar because it serves as intermediary for coordinating the interactions among groups of objects but it is different because its main aim is to decrease the complexity of interactions. The second is similar because it adapts the interfaces of the objects while it differs because our mediator is not just an interface translator.

In the Web services context several works have introduced basic pattern mismatches and the corresponding template solutions to help the developers to compose mediators [2, 5, 30, 31]. The Web Services research community has been also studying solutions to the automatic mediation of business processes in recent years [12, 13, 7, 9]. A lot of work has been also devoted to behavioral adaptation that is related to our work among which, for instance, [3]. It proposes a matching approach based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioral descriptions. Our matching, as sketched in Section 3, is driven by the ontology and is better described in [18] where the theory underlying our approach is described at a high level and in [17] where a more detailed version of the theory can be found. The work [25] presents an algebra over services behavioral interfaces to solve six mismatches and a visual notation for interface mapping. The proposed algebra describes with a different formalism solutions similar to our basic patterns LTSs and this can be of inspiration for us in the direction of the reasoning. Finally in [29] the authors propose an approach for software adaptation which takes as input components behavioral interfaces and adaptation contracts and automatically builds an adaptor such that the composed system is deadlock-free. The adaptor generation is also tool supported. Similarly to us, they also solve some mismatches. A difference, instead, is that our aim is to achieve communication without adding extra constraints while they say that their main goal is to ensure deadlock freedom and addresses system-wide adaptation specified through policies and properties.

8 Conclusion

The Ubiquitous environment, embedding a big number of heterogeneous system's components, puts forward an ever growing need of mediation entities for component's interoperability purpose. The challenge is to embed *mediators* components able to solve the component's behavioral discrepancies, into the system architecture allowing components to coordinate and communicate. Moreover, due to the continuous evolution of such environments, recently has also emerged the issue of dynamic and on the fly solutions to behavioral diversities letting hence arise the challenge of automatic approaches to cope whit this problem.

To respond to these two challenges, we first illustrated the Mediating Connector Architectural Pattern which, encapsulating the necessary support, is the key enabler for the communication between mismatching components. Indeed, it solves the interoperability problems between heterogeneous and potentially compatible components. With respect to the second challenge, we proposed an automatic pattern based approach detailing, in particular, a set of Basic Mediator Patterns, including basic mismatches and respective solutions, which represent the basic building blocks on which an automatic approach can build upon.

As future works, we intend to: define a theoretical compositional strategy to allow reasoning on mismatches and to build the mediating connector behavior. Moreover we also aim at providing the "concrete" Basic Mediator Patterns, i.e., the skeleton code corresponding to the "abstract" ones presented in this work and present the actual code for the component's behavior decomposition and the mediating connector behavior building.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein. A Pattern Language, vol. 2 of Center for Environmental Structure Series. Oxford University Press, New York, NY, 1977.

[2] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani, Developing adapters for web services integration In CAiSE, Porto, Portugal. Springer Verlag, 2005, pp. 415-429.

[3] Motahari-Nezhad, H.R., Xu Guang Y. and Benatallah, B., "Protocol-Aware Matching of Web Service Interfaces for Adapter Development" In WWW'10, pp. 731-740. ACM, New York, USA.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons, August 1996.

[5] E. Cimpian and A. Mocan, Wsmx process mediation based on choreographies. in Business Process Management Workshops, C. Bussler and A. Haller, Eds., vol. 3812, 2005, pp. 130143.

[6] M. Weiser, The computer for the 21st century, Scientific American, Sep. 1991.

[7] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In WWW '07, pages 993-1002, New York, NY, USA, 2007. ACM.

[8] R. Spalazzese, P. Inverardi. Mediating Connector Patterns for Components Interoperability. In ECSA2010 - European Conference on Software Architecture, LNCS 6285, pp.335-343. Springer-Verlag Berlin Heidelberg 2010.

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

[10] CONNECT European project, http://connect-forever.eu/.

[11] V. Issarny, B. Steffen, B. Jonsson, G. Blair, P. Grace, M. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In ICECCS, Postdam Germany, 2009.

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

[13] R. Vaculin, R. Neruda, and K. P. Sycara. An Agent for Asymmetric Process Mediation in Open Environments. In R. Kowalczyk, M. N. Huhns, M. Klusch, Z. Maamar, and Q. B. Vo, eds, SOCASE, volume 5006 of LNCS, pages 104-117. Springer, 2008.

[14] R. Kumar, S. Nelvagal, and S. I. Marcus. A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems, 7(3):295-315, 1997.

[15] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In ICSE '03, pages 374-384, Washington, DC, USA, 2003. IEEE Computer Society.

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

[17] Inverardi, P., Issarny, V., Spalazzese, R.: A theory of mediators for eternal connectors. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236250. Springer, Heidelberg (2010)

[18] R. Spalazzese, P. Inverardi, and V. Issarny. Towards a formalization of mediating connectors for on the fly interoperability. In WICSA/ECSA 2009, pages 345-348, 2009.

[19] M. A. Motoyama and G. Varghese. Crosstalk: scalably interconnecting instant messaging networks. In WOSN '09, pages 61-68, New York, NY, USA, 2009. ACM.

[20] P. Avgeriou, U. Zdun. Architectural Patterns Revisited A Pattern Language. In EuroPLoP 2005, pages 139, Irsee, Germany, 2005.

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

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

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

[24] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In ICSE'07, pp. 54-64, IEEE Computer Society, Washington, DC, USA.

[25] M. Dumas, M. Spork, K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation". In S. Dustdar, J.L. Fiadeiro, and A. Sheth (Eds.): BPM 2006, LNCS 4102, pp. 6580, 2006. Springer-Verlag Berlin Heidelberg 2006

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

[27] N. F. Noy, Semantic integration: a survey of ontology-based approaches, SIGMOD Rec., vol. 33, no. 4, pp. 6570, 2004.

[28] A. Bertolino, P. Inverardi, P. Pelliccione, M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In Proc.ESEC/FSE. pp. 141-150, 2009.

[29] C. Canal, P. Poizat, G. Salaün, "Model-Based Adaptation of Behavioral Mismatching Components," IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 546-563, July/Aug. 2008.

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

[31] F. Jiang, Y. Fan, and X. Zhang. Rule-based automatic generation of mediator patterns for service composition mismatches. In of GPC-WORKSHOPS '08, pages 3-8, Washington, DC, USA, 2008. IEEE Computer Society.