The Many Facets of Mediation: A Requirements-driven Approach for Trading-off Mediation Solutions

Book Section

The Many Facets of Mediation

A Requirements-driven Approach for Trading-off Mediation Solutions

Amel Bennaceur[1] and Bashar Nuseibeh[1,2]

[1] The Open University, Milton Keynes, UK
firstname.lastname@open.ac.uk
[2] Lero - The Irish Software Research Centre, Limerick, Ireland
firstname.lastname@lero.ie

Abstract. Mediation aims at enabling dynamic composition of multiple components by making them interact successfully in order to satisfy given requirements. Through dynamic composition, software systems can adapt their structure and behaviour in dynamic and heterogeneous environments such as ubiquitous computing environments. This paper provides a review of existing mediation approaches and their key characteristics and limitations. We claim that only a multifaceted approach that brings together and enhances the solutions of mediation from different perspectives is viable in the long term. We discuss how requirements can help identify synergies and trade-offs between these approaches and drive the selection of the appropriate mediation solution. We also highlight the open issues and future research directions in the area.

Keywords: Mediator synthesis, Requirements, Architectural mismatches.

1 Introduction

To software developers, life may sometimes seem like a scene from "Modern Times" where Charlie Chaplin is labouring away at an assembly line, frantically tightening bolts over and over again. Modern software systems are increasingly built by assembling, and re-assembling, existing components —possibly distributed among many devices— so as to create innovative services. Since the components of a software system are often designed and implemented independently, software developers spend a lot of time, and effort, adding pieces of code so as to allow these components to work together and satisfy the requirements of the software system. The rapid pace of technological change combined with the increasing demands for high-quality software in reduced time and at lower cost, may overwhelm developers who have to deal with a multitude of details just to make components work together. Besides being a complex and error-prone task, enabling independently-developed components to work together is both daunting and tedious. Developers should be free to spend more time creating new services and designing innovative software systems and less time tightening and re-tightening bolts. Therefore, we must enable independently-developed software

components to work together, if need be, despite the many differences in their implementations.

Middleware provides an abstraction that facilitates the communication and coordination of distributed components despite the heterogeneity of the underlying platforms, operating systems, and programming languages. However, middleware also defines specific message formats and coordination models, which makes it difficult (or even impossible) for applications using different middleware solutions to interoperate. For example, SOAP-based clients developed using Java and deployed on Mac can seamlessly access a SOAP-based Web Service developed using ASP.NET and deployed on a Windows server. However, a SOAP-based client cannot access a RESTful Web Service [19]. Furthermore, the evolving application requirements lead to a continuous update of existing middleware tools and the emergence of new approaches. For example, SOAP has long been the protocol of choice to interface Web services but RESTful Web services are somehow prevailing nowadays. As a result, application developers have to juggle with a myriad of technologies and tools, and include *ad hoc* glue code whenever it is necessary to integrate applications implemented using different middleware.

To make heterogeneous components work together, without modifying them, intermediary software entities, called *mediators*, are used [54]. Mediators achieve interoperability by reconciling the differences in the implementations of the components involved. Hence, mediators enable compositional adaptation [37], which aims to change the behaviour and the structure of a system to make it better fit its environment. Designing and implementing mediators requires dealing with many concerns: (i) coordination of the behaviours of the components so as to guarantee their correct interaction (e.g., absence of deadlocks), (ii) data translation so as to ensure meaningful information exchange between the components, and in the case of distributed components (iii) communication between the components so as to address the issues inherent in their distribution across the network (e.g., concurrency and fault tolerance).

Over the years, mediator synthesis has been the subject of a great deal of work, both theoretical and practical. First, to understand and formalise architectural connection and mismatches, then to synthesise mediators to solve these mismatches with an increasing shift towards runtime. While mediation has been a long-researched topic, the advent of mobile and ubiquitous computing technology emphasises the need for more dynamic solutions to mediation, and compositional adaptation in general. These solutions are not only applicable at design time but also at runtime. For example, consider one representative application domain, that of emergency management, as illustrated by the European Programme for the establishment of a European capacity for Earth Observation, GMES[3]. GMES gives a special interest to the support of emergency situations (e.g., forest fire) across different European countries. Indeed, each country defines an emergency management system that encompasses multiple components that are autonomous, designed and implemented independently, and do not obey any central control or administration. Nonetheless, there are incentives for these com-

[3] Global Monitoring for Environment and Security –`http://www.gmes.info/`

ponents to be composed and collaborate in emergency situations. GMES makes a strong case of the need for solutions to enable multiple, and most likely heterogeneous, components to collaborate in order to perform the different tasks necessary for decision making. These tasks include collecting weather information, locating the agents involved, and monitoring the environment. In this context, the synthesis of mediators enables the dynamic composition of heterogeneous components whose interaction was unforeseen at design time.

In this paper we present a review of current research in mediation, presented from the perspective of its underpinning fields: *software architecture, middleware, formal methods and Semantic Web*. Mediator synthesis is a complex challenge that can only be solved by appropriately combining different techniques and perspectives. These techniques include formal approaches for the synthesis of mediators with the support of ontology-based reasoning so as to automate the synthesis, together with middleware solutions to realise and execute these mediators. While these different techniques focus on *How* to synthesise mediators that make components interact in order to achieve a single property, requirements primarily focus is on *Why* components should be mediated and for which properties. Therefore, requirements can drive the selection of the appropriate method for synthesising mediators. In this chapter, we present a requirements-driven approach for managing trade-offs between the different solutions to mediation in order to choose the appropriate one.

This chapter is structured as follows. Section 2 gives an overview of the different perspectives on mediator synthesis, which are then detailed in the following sections (3 to 6). Section 7 proposes a framework that unifies the different solutions. Section 8 identifies the opportunities and challenges for using requirements to drive mediation. Finally, Section 9 concludes the chapter.

2 The Different Perspectives on Mediation

In this section we present the different approaches to mediation seen from the perspective of its underpinning fields: software architecture, middleware, formal methods and Semantic Web. Fig. 1 depicts, for each perspective, the specific focus and the main technique used as well as how mediators are considered:

❶ Software architecture focuses on *composition*: several software entities are put together to build a system and define its *structure* as a whole [47]. Interaction between components is abstractly described using software connectors. In other words, connectors model the exchange of information between components and the coordination of their behaviours. Hence, mediators can be conveniently represented as connectors.

❷ Middleware provides an abstraction that facilitates communication and coordination between components in *distributed* systems. It naturally follows that middleware plays a crucial role in the *implementation* of connectors [38].

❸ Formal methods are mathematically-based languages, techniques, and tools for specifying and verifying hardware and software systems [15]. Formal

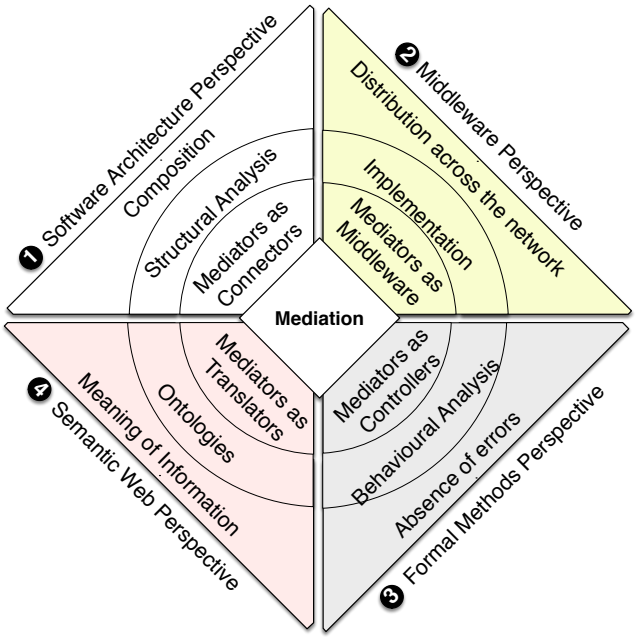

Fig. 1. The different perspectives on mediation

methods focus on the *behaviour* of software systems, which they rigorously analyse in order to reveal potential inconsistencies, ambiguities, and incompleteness. In other words, formal methods help to verify the *absence of execution errors* in software systems. Once potential execution errors (a.k.a. mismatches) are detected, they can be solved by introducing *controllers* that force the components to coordinate their behaviours correctly.

❹ The Semantic Web is an extension of the Web in which information is given well-defined *meaning*, better enabling computers and people to work in cooperation [4]. Ontologies play a key role in the Semantic Web by formally representing shared knowledge about a domain of discourse as a set of concepts, and the relationships between these concepts [24]. Ontologies have been extensively used to automate the reasoning about the information exchanged between software components, especially in ubiquitous computing environments, so as to infer the translations necessary to reconcile the differences in the syntax of this information [36].

We detail the techniques for mediation from each perspective in the following. We first adopt a software architecture perspective to present the concepts underpinning mediator synthesis. Next, we concentrate on middleware for the implementation of and deployment of mediators. Then, we describe formal solutions that analyse the behaviours of components in order to synthesise the me-

diator that guarantees that they can interact without errors. Finally, we present solutions based on ontologies so as to represent and reason about the meaning of the information exchanged between components at runtime and automatically synthesise mediators. Note that these perspectives are not orthogonal and some techniques can be classified in more than one perspective.

3 The Software Architecture Perspective: Mediators as Connectors

Software architecture abstractly describes the structure of software systems in terms of components and connectors [47]. A component encapsulates some functionality to which it restricts access via an explicit interface [51]. To achieve its functionality, the component interacts with the environment and other components, that is the component's behaviour. A connector regulates interactions between components [51].

One critical issue for software architecture is the design and implementation of the connectors that permit the various software components to work together properly. However, when composing two, or more, software components to form a system and those components make conflicting assumptions about their environment, *architectural mismatches* occur [20]. These assumptions relate to: (i) the interfaces and behaviours of the components involved, (ii) the behaviours and implementations of the connectors used, and (iii) the operating systems and the hardware of the devices on top of which the components are deployed.

Mediation aims to solve architectural mismatches by reconciling the conflicting assumptions that the components make about their environment. To solve the differences between the interfaces of components, the mediator must translate the actions required by each of them into actions provided by the other. Note that the mediator facilitates interaction —it is a connector— but does not provide any action itself since it does not encapsulate computation. To solve the differences between the behaviours of components, the mediator must coordinate the exchange of information between these components by controlling which action should be delivered to which component at what time. To solve the differences between the behaviours and implementations of connectors, the mediator must provide a concrete solution to coordinate the interaction patterns of these connectors acting as middleware, which not only makes the application agnostic to the operating systems, but also to the middleware used to implement other connectors.

Connector synthesis. It is not always possible to find an existing connector for managing interaction between heterogeneous components and it is difficult and time consuming to design and implement a new connector from scratch, especially if the components already exist and are implemented using different middleware solutions [38]. Compositional approaches for connector construction facilitate the development of mediators by reusing existing connector instances.

Spitznagel and Garlan [49] introduce a set of transformation patterns (e.g., data translation and action aggregation), which a developer can apply to basic connectors (e.g., RPC or data stream) in order to construct more complex connectors. The authors use the approach to enhance the reliability of component interactions, but state that this approach can also be used to construct mediators that solve architectural mismatches. Each transformation pattern is given a formal definition, which allows the verification of the properties of the resulting connectors. As developers are responsible for defining the transformation patterns, they must specify both the necessary translations and behavioural coordination that must be performed by the mediator, but they can easily verify that the mediator produced ensures that the interaction between components is free from deadlocks. The approach is also equipped with a tool that facilitates the implementation of mediators by reusing and composing the implementation of existing connectors, assuming existing connectors were implemented using the same middleware.

Inverardi and Tivoli [27] define an approach to compute a mediator that composes a set of pre-defined patterns in order to guarantee that the interaction of components is deadlock-free. These patterns represent simple mechanisms that the mediator executes to solve differences between the interfaces or behaviours of components and consist of: (i) renaming an action, (ii) translating one action into a sequence of actions, (iii) translating a sequence of actions into one action, (iv) re-ordering sequences of actions, (v) dropping an action, and (vi) introducing a new action. This last pattern has to be taken with reserve as it implies that the mediator is able to produce an action. The mediator either only replays the action or it can perform extra computation; the latter case being beyond interoperability achievement. However, the specification of the patterns to be used must still be done by the developers. Indeed, developers specify the necessary translations based on which the approach synthesises the mediator that coordinates the behaviours of the components. Furthermore, the implementation of the resulting mediator is completely left up to the developer as the mediator is generated from scratch without reusing existing connector implementations.

Even though these compositional solutions facilitate the development of mediators, they are only applicable at design time. By requiring the intervention of the developer to specify the patterns necessary for the creation of mediators, they cannot cope with the increasing ubiquity and complexity of modern software systems together with the high demand for runtime support.

Connector synthesis in dynamic environments. Building mediators is already a difficult task when the developer provides the necessary translations. It is even more difficult when the mediators have to be synthesised and deployed dynamically as components are discovered and composed at runtime.

Chang *et al.* [13] define a framework that allows component developers to define connectors, called *healing connectors*, to recover from common failures of the component. The healing connectors enable the component to operate in environments that do not verify the assumptions made during the design and

implementation of this component. At runtime, whenever an exception rises due to the misuse of a component, the framework deploys, on the fly, the corresponding healing connector. The framework also maintains a log of the exceptions in order to help developers create new healing connectors. Denaro *et al.* [17] apply the same approach to detect and repair disparities in different implementations of standard Web 2.0 APIs. The healing connectors are not defined by the developers but are included in a centralised catalogue that inventories the common errors that may occur when the API is used.

However, the proposed solutions only react to errors during the execution of a single action and do not consider the behaviours of the components. Hence, they solve architectural mismatches which are due to conflicting assumptions regarding the components' interfaces, but not due to conflicting assumptions about the components' behaviours. Furthermore, healing connectors act as translators for the case of common misuse based on the experience of developers and are not able to deal with unforeseen interactions. The implicit knowledge used by the developer to specify the translator should be modelled explicitly in order to allow computers to reason about the information exchanged by the components and infer the translations automatically.

Analysis. Considering mediation from a software architecture perspective allows us to define the foundational concepts for the formal description, synthesis, and implementation of mediators. Mediators are connectors that enable components to work together by translating the actions of their interfaces and coordinating their behaviours. In ubiquitous computing environments, mediators must be generated on the fly to deal with the high-degree of dynamism inherent in these environments. In the following, we first consider the middleware solutions that facilitate the implementation of mediators by compensating for the differences at the middleware layer. Then, we present the formal solutions to synthesising mediators that coordinate the behaviours of functionally-compatible components in order to guarantee their successful interaction. Finally, we consider semantics-based solutions to infer the translations necessary for meaningful exchange of information between components and enable the synthesis of mediators at runtime.

4 The Middleware Perspective: Mediators as Middleware

Middleware makes components work together by hiding the differences in hardware and operating systems, as depicted in Fig. 2. Middleware facilitates communication and coordination between components in distributed systems by defining [28]: (i) an Interface Description Language (IDL) for specifying the interfaces of components and the associated operations, and data types, (ii) a discovery protocol to address and locate the components that are available in the environment, and (iii) an interaction protocol that coordinates the behaviour of different components and enables them to collaborate. While middleware solutions and implementations define diverse IDLs and message formats, their

interaction protocols follow comparably few interaction patterns, a.k.a., communication paradigms/types [16] or coordination models/paradigms [28]. An interaction pattern defines the rules to coordinate the behaviours of the components. In Mehta *et al.* connector classification [39], these interaction patterns match with the connector types that provide communication and coordination services. The major interaction patterns are: Remote Procedure Call (RPC), Distributed Shared Memory (DSM), and Publish/Subscribe [16].

RPC represents the most common interaction pattern in distributed systems. This approach directly and elegantly supports client/server interactions with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally. The interaction is supported by a pairwise exchange of messages from the client to the server and then from the server back to the client, with the first message containing the operation to be executed at the server and associated arguments and the second message containing any result of the operation. To interact according to RPC, the client and the server must agree on the format of the messages they exchange as well as the encoding of the data, which represent the arguments and results, enclosed in these messages. An RPC-based middleware hides the encoding together with the decoding of arguments and results as well as the passing of messages using communication modules, *stubs*, that permit the client and server to use the operations as if they were local. RPC-based middleware solutions are often associated with libraries to generate, either at compile time or runtime, the client and server stubs based on the interface definition. The strict request-reply message exchange is unnecessary when there is no result to return. RPC middleware solutions may also provide facilities for what are called asynchronous RPCs, by which a client immediately continues its execution after issuing the RPC request.

While RPC allows developers to invoke operations as if they were available locally, DSM provides developers with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces. DSM is in general less appropriate for client/server interactions, where clients usually access server-held resources using an explicit interface (for reasons of modularity and protection). Still, servers can provide DSM that is shared between clients. A DSM-based middleware enables components to read and write data in the shared memory, regardless of the exact location of the data. Nevertheless, the structure of the shared data is defined at the application layer and the middleware does not provide any guarantee about when data is made available and how long it will reside in the shared memory. In other words, the synchronisation between the readers and writers also needs to be managed at the application layer.

Many applications require the dissemination of information or items of interest from a large number of producers to a similarly large number of consumers. Publish/Subscribe middleware solutions provide an intermediary service, a *broker*, that efficiently ensures that information generated by producers is delivered to the consumers that want to receive it. In other words, publish/subscribe middleware solutions (sometimes also called distributed event-based middleware)

allow subscribers to register their interest in an event, or a pattern of events, and ensure that they are asynchronously notified of events generated by publishers. The task of the publish/subscribe middleware is to match subscriptions against published events and ensure the correct delivery of event notifications. A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many interaction pattern. The expressiveness of publish/subscribe middleware solutions is determined by the type of event subscriptions they support: either subscriptions are made using specific topics (also referred to as subjects) which the events belong to, or based on the content of the event.

Traditionally, middleware promotes the use of a single technology based on which all components are built, which can be based on RPC (e.g., RMI and RPC SOAP), DSM (e.g., Linda and LIME), or Publish/Subscribe (e.g., JMS and AMQP). However, given the diversity of modern software systems that need to be dealt with, ranging from small-scale sensors to large-scale Internet applications, there is no one-size-fits-all middleware capable of coping with them all [6]. As a result, new middleware solutions have been proposed to enable interaction across middleware and hence facilitate the implementation of mediators between independently-developed components that feature differences at both the application and middleware layers. We first present solutions based on the definition of middleware that provides developers with an abstraction which allows them to build components that are able to interact using different middleware solutions, i.e., *universal middleware*. We then consider solutions to directly translate messages from one middleware to the other, i.e., *middleware bridges*. Finally, we consider solutions to translate between different middleware solutions using an intermediary model or infrastructure, i.e., *service buses*.

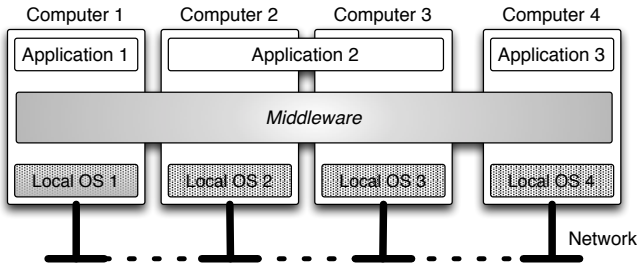

Fig. 2. Middleware [50]

Universal middleware. Universal middleware solutions provide the developer with an abstraction that masks the differences that may exist at the middleware layer. Solutions include polymorphic middleware such as PolyORB [53] and reflective middleware such as ReMMoC [23].

PolyORB [53] is a middleware solution that decouples the interaction pattern used to implement the application from the middleware used for the actual achievement of this interaction. First, PolyORB supports several interaction patterns, called *application personalities*, based on which applications can be developed. Second, PolyORB supports different communication protocols called *protocol personalities*, e.g., SOAP and GIOP. The relation between the application and protocol personalities is handled via an intermediary protocol into which any application personality can be translated and which can be translated into all protocol personalities. Before deploying the component, it is configured with the appropriate personalities. Hence, it is not possible to select the appropriate protocol personality dynamically according to the running environment.

ReMMoC (Reflective Middleware for Mobile Computing) [23] is a reflective middleware solution that provides a WSDL-based interface to develop components. ReMMoC implements a set of plugins to transform the primitives of the WSDL interface into calls to other middleware technologies, in particular SOAP, CORBA, and STEAM (a publish/subscribe middleware). At runtime, a component implemented using ReMMoC can discover and interact with components implemented using different middleware solutions by dynamically loading the necessary plugin.

An approach based on universal middleware has many flaws. First, it cannot be applied to legacy components, as it requires at least one of the interacting components to be developed using the universal middleware. Second, the universal middleware must support any possible middleware and hence requires continual maintenance in order to cope with the evolution of middleware solutions or the emergence of new ones.

Middleware bridges. To deal with interoperability between existing components, the most straightforward solution is to develop a middleware solution that implements direct translation between the messages of two middleware solutions. The middleware bridge takes messages from one middleware in a specific format and then marshals them to the format of the other middleware.

There exist several examples of middleware bridges: OrbixCOMet[4] is a middleware bridge between DCOM and CORBA and SOAP2CORBA[5] ensures interoperability between SOAP and CORBA in both directions. However, the implementation of middleware bridges is a complex task: developers have to deal with a lot of details involving the format of the messages used by each middleware and their correlation; therefore, developers must have a thorough understanding of the middleware at hand. As a result, solutions that help developers define middleware bridges have emerged. These solutions consist in defining a framework whereby the developer provides a declarative specification of the message translation between middleware, based on which the actual transformations are

[4] `http://documentation.progress.com/output/Iona/orbix/gen3/33/html/orbixcomet33_pguide/`

[5] `http://soap2corba.sourceforge.net/`

computed. z2z [8] introduces a domain-specific language to describe the message format and the communication protocol of each middleware as well as the translation logic to make them work together, and then generates the corresponding bridge. The approach has several benefits. First, it increases the level of reusability as the developer can use the individual specifications of middleware to develop different bridges. Second, the developer does not have to deal with all the message fields since z2z is able to complete default and optional fields automatically. Finally, z2z verifies that all the required fields of a message have been treated before sending it. However, the bridge cannot be modified at runtime.

Starlink [7] uses the domain-specific models defined by z2z to specify bridges, but it deploys and interprets them at runtime. More specifically, Starlink uses the message specification associated with each middleware to generate a *parser*, which is able to process the messages sent using this middleware into an *abstract message*, and a *composer*, which is able to produce the appropriate middleware message out of an abstract message. In other words, parsers and composers mask the differences between middleware through the concept of abstract messages. The translation logic specifies how to convert the abstract messages of one middleware into abstract messages of the other middleware. This approach decouples the detailed specification of the middleware, which is used to generate the corresponding parsers and composers, from the abstract specification of the translations between middleware solutions.

Summing up, middleware bridges provide a transparent solution to interoperability but are impractical in the long term given the development effort necessary to implement or specify the translation between middleware solutions. Furthermore, in the case of middleware based on different interaction patterns, this translation may become unfeasible in all situations, for example, if one middleware is based on asynchronous communication while the other relies on synchronous communication.

Service buses. Like middleware bridges, service buses enable existing components implemented using different middleware to exchange messages transparently, but unlike middleware bridges, the translation between messages is performed through an intermediary representation. This representation can be an abstract proprietary protocol, as is the case with middleware buses, or a message-oriented abstraction layer, as is the case with enterprise service buses.

Georgantas *et al.* [21] define an approach where the developer specifies a set of semantic events common to different middleware. Then, each middleware is associated with a parser that processes the messages of this middleware to produce a semantic event, and a composer that generates a middleware message based on a semantic event. Parsers and composers of different middleware then synchronise based on shared semantic events. For example, to achieve interoperability between SOAP and CORBA, developers define the request and response events. Then, parsers and composers are created per protocol: a SOAP parser triggers a request (respectively response) event upon the reception of a

SOAP request (respectively response) and a SOAP composer produces a SOAP request (respectively response) out of a request event (respectively response). The same is true for CORBA parsers and composers. Hence, when a SOAP request is received, the SOAP parser triggers a request event, which the CORBA composer intercepts and transforms into a CORBA request. Once the CORBA response has been returned, the CORBA parser triggers a response event, which the SOAP composer intercepts and transforms into a SOAP response. However, this approach is inapplicable for middleware based on different interaction patterns since it is also necessary to coordinate the message exchange as well as the translation between messages. Furthermore, the approach does not provide any support for the specification or implementation of application-level mediators.

Enterprise Service Buses (ESBs) represent the most mature and widespread solution to enable components using different middleware to interoperate, as is shown by the large number of available industrial implementations, e.g., Oracle Service Bus[6] and IBM WebSphere Enterprise Service Bus[7]. An ESB [40] is an open standard, message-based middleware solution that facilitates the interactions of disparate distributed applications and services. ESBs generally include built-in conversion across standard middleware technologies (e.g., SOAP, JMS) and provide a set of predefined patterns that can be used to create customised mediators.

However, ESBs takes an enterprise perspective, where interactions between components are planned and long-lived. Hence, the solutions are typically restricted to a set of known middleware standards, and the development effort required to extend them for new protocols or to specify mediators is significant. They are not well suited to situations where interactions must be solved on the fly as in ubiquitous computing environments, which involve short-lived interactions and unforeseen compositions.

Analysis. There exist many middleware solutions to enable components that feature differences at the middleware layer to interact successfully. However, while the implementation of new middleware might be sufficient to deal with the differences at the middleware layer, it is insufficient to deal with differences at the application layer. First, even applications developed using the same middleware are not guaranteed to work together so long as there are differences in their interfaces and behaviours. This is, for example, the case of interoperability in Web Services [41]. Even though both services and clients use SOAP middleware, the differences between their interfaces, which include differences in the operation names, input/output message names and types, the granularity of operations, and the order in which these operations are invoked (or expected to be invoked) hamper independently-developed clients and Web Services from working together. Given the countless number of potential cases where a mediator is necessary, any static solution is doomed to fail. We need to generate mediators

[6] `http://www.oracle.com/technetwork/middleware/service-bus/`
[7] `http://www-01.ibm.com/software/integration/wsesb/`

automatically. Second, while in the case of middleware obeying the same interaction pattern, it suffices to translate the messages sent using one middleware into messages expected by the other middleware, when middleware solutions follow different interaction patterns, e.g., a shared memory and publish/subscribe, the differences can only be solved by considering the characteristics of the applications [12]. Hence, it is necessary to define solutions that are able to reason about the characteristics of applications automatically in order to synthesise the mediator that reconciles the differences between component implementations and enables them to interoperate. In the following section, we present solutions that analyse the behaviours of the components and semi-automatically generate the appropriate mediator that enables their correct interaction.

5 The Formal Methods Perspective: Mediators as Controllers

Formal methods aim to relieve developers of the burden of designing or specifying mediators, with a special focus on coordinating the behaviours of the components so as to guarantee their correct interaction. Correct interaction may be specified as: (i) the ability of the components to coordinate their behaviours in order to achieve the requirements of the composed system, or (ii) the ability to preserve the meaning of the information exchanged between the components and guarantee that the composed system is free from deadlocks.

Controller synthesis using a specification of the composed system. The successful interaction of components results in a composed software system that meets given requirements. By enabling components to interact with each other, mediators can be seen as the missing behaviour necessary to realise a specification of the composed system *Goal*.

Quotient. Calvert and Lam [10] formulate mediator synthesis as the problem of finding *quotient*. In a similar way to division and product in arithmetics, quotient can be regarded as the adjoint (roughly "inverse") of parallel composition. Given a specification for a system S, together with a component's behaviour P, the quotient yields the behaviour Q such that $P \| Q$ satisfies S. Applied to mediator synthesis, the mediator is the quotient of the specification of the composed system *Goal* and the parallel composition of the components' behaviours. The authors assume *Goal* to be deterministic and synthesise M by first building the set of all possible coordinations of the actions of the components' interfaces, and then keeping only those that satisfy *Goal*.

Even though the approach can, in theory, always produce a mediator if one exists, it is clear from the algorithm that it is computationally very expensive as it requires exploring all possible traces over the set of actions of both *Goal* and M. Furthermore, it assumes that the same actions are used to define the specification of the composed system as well as the components' behaviours.

Planning. Similarly to quotient computation, the planning-based approach defined by Bertoli *et al.* [5] builds the mediator by identifying among all possible interactions with the components, only those that satisfy *Goal*. Nevertheless, they optimise the search by using a heuristic in order to explore only the interactions that are likely to satisfy *Goal* and use a planning algorithm in order to calculate the traces of the mediator more efficiently.

Control theory. Gierds *et al.* [22] formulate mediator synthesis in terms of controller synthesis. Besides the components' behaviours and the specification of the composed system, they also require the definition of a set of translation patterns between the actions of the components. They create a component whose behaviour E is extracted from the specified translation patterns: E represents the behaviour of a component able to execute the translation patterns in any order. Then, they use available tools for controller synthesis to generate a controller C for the composition $P_1 \| P_2 \| E$ to satisfy *Goal*. Finally, they compose the behaviour of the controller together with the behaviour of the translation component to obtain the mediator, i.e., $M = E \| C$.

Summing up, solutions to mediator synthesis based on quotient computation, planning or controller synthesis are guaranteed to find the mediator if it exists and state its non existence otherwise. However, they require the user to have an intuitive understanding of the behaviour of the composed system, which can only emerge through the correct interaction of its components. This might be a reasonable assumption when developing a software system by integrating several components, but it is unreasonable to require such understanding from regular users who only seek to interact with the services in their environment, as is the case in ubiquitous computing environments.

Controller synthesis using a partial specification. The solutions proposed in the following assume that a specification of the correspondence between the actions of the components' interfaces is available and use it to coordinate the components' behaviours in order to guarantee that their interaction is free from deadlocks. This correspondence defines the translations that the mediator must perform in order to reconcile the differences between the components' interfaces. Therefore, we refer to the specification of these correspondences as partial specifications of the mediator.

Projection. Lam [31] defines an approach for the synthesis of mediators based on the technique of *projections*. A projection of a component's behaviour P, noted $Proj[P]$ is performed by aggregating some of its states, which induces the definition of an equivalence relation on the actions of the component's interface. Two actions are equivalent if they cause identical state change in $Proj[P]$ while actions that do not cause any state change are not represented in $Proj[P]$. Hence, the projection can be seen as applying relabelling and hiding functions to P.

If one can define a *useful* common projection of the behaviours of the components, then a stateless mediator M can be synthesised. Useful means that the

common projection defines a behaviour to achieve some functionality of interest. The common projection can be seen as the lowest common denominator of the behaviours of the components. The definition of the common projection is the responsibility of the developer. The synthesised stateless mediator simply transforms an action required by one component into an action provided by the other component if they cause identical state change in the common projection, and ignores the actions that do not cause any state change. However, this stateless mediator is able to deal with only one-to-one correspondences between actions. Furthermore, no systematic approach for the definition of the common projection is proposed, it solely depends on developers and their understanding of the components' behaviours.

Interface mapping. Yellin and Strom [55] define a synthesis algorithm that, besides the behaviours of the components, must be given an interface mapping S, which specifies the correspondence between the actions of the components' interfaces. The interface mapping is required to be complete and non-ambiguous. An interface mapping is complete if for every required action of one component, there corresponds a provided action from the other component. It is non-ambiguous if for every required action of one component, there corresponds at most one provided action from the other component. Each correspondence in the interface mapping defines an ordering constraint between the required and provided actions. The synthesis algorithm constructs a mediator in two main phases. During the first phase, an initial process A is created which represents all possible coordinations of components' behaviours that verify the ordering constraints imposed by the interface mapping. In the second phase, any execution in A leading to a deadlock is removed. As a result of the second phase, either A is empty, in which case the mediator does not exist, or it is a valid mediator M.

Model checking. While interface mapping only specifies one-to-one correspondences between actions, there often exist more elaborate correspondences relating them. In the general case, a sequence of actions of one component may be translated into another sequence of actions of the other component. To specify complex correspondences, Mateescu *et al.* [35] use an *adaptation contract*, which is an LTS S whose alphabet is a vector composed of the actions of the components' interfaces. The authors then construct the mediator by selecting among all possible executions of the composed system C only those that do not lead to deadlocks. Instead of constructing C then removing the erroneous executions, they use on-the-fly model checking to prune, as early as possible, the executions leading to deadlocks.

Semi-automated mapping generation. Nezhad *et al.* [42, 43] define a semi-automated approach to the synthesis of mediators which, rather than considering that the correspondences between the actions of the components are provided, define a series of heuristics to facilitate their computation. First, they focus on the syntax, expressed using XML schema, of the data embedded in the actions. They use existing XML schema matching techniques to evaluate the degree of

similarity between sequences of actions in the components' interfaces. Then, they update this similarity based on the first position at which the actions can appear in the behaviours of the components: the similarity score of required and provided actions increases if they are at the same position. The last heuristic consists in selecting the pair of actions with the highest degree of similarity according to the matching of their XML schema and then updating the similarity scores of the other pairs of actions according to their positions relative to the selected pair of actions. The same pair of actions is never selected twice so that the heuristic is guaranteed to terminate. Once the correspondences between actions have been computed, the behaviours of the two components are simultaneously explored in order to identify possible deadlocks. The user is presented with the deadlocks that may occur and has to figure out the appropriate translations that may solve them. The algorithm cannot apply the mapping directly as there is no guarantee that even the actions with the highest similarity score have the same meaning.

Analysis. Formal methods enable a rigorous analysis of components' behaviours in order to synthesise the mediator that coordinates the components' behaviours appropriately. Nevertheless, besides the description of components' behaviours, the synthesis of mediators using formal methods also requires the specification of a single propertiy of the composed systems or the correspondence between actions. The definition of the correspondences between the actions of components' interfaces may be error-prone given the size and the number of parameters of the interfaces involved. For example, the Amazon Web Service[8] includes 23 operations and no less than 72 data type definitions and eBay[9] contains more than 156 operations. Given all possible combinations, methods that automatically compute these correspondences are necessary.

6 The Semantic Web Perspective: Mediators as Translators

When the components are dynamically discovered, and interact spontaneously, as is the case in ubiquitous computing environments, the correspondences between the actions of components' interfaces must also be elicited at runtime. To do so, the meaning of these actions and their relations must be made explicit in order to allow their automated analysis.

Therefore, the Semantic Web [4] promotes the view that Web resources are augmented with machine-processable metadata expressing their meaning. This vision is supported by ontologies, which provide a machine-processable means to represent and automatically reason about the meaning of data based on the shared understanding of the domain [25]. By relying on ontologies, Semantic Web Services improve the discovery, composition, and mediation of Web Services.

[8] `http://soap.amazon.com/schemas2/AmazonWebServices.wsdl`
[9] `http://developer.ebay.com/webservices/latest/ebaysvc.wsdl`

Semantic Web Services. Web Services are processes that expose their interfaces to the Web so that users can invoke them. Semantic Web Services provide a richer and more precise way to describe the services through the use of knowledge representation languages and ontologies. The aim is to facilitate the service discovery and composition by exploiting knowledge explicitly encoded in the ontology rather than trying to guess the meaning encoded in the schemas, as is the case with XML schemas for example [48]. Major efforts for modelling and using Semantic Web Services include OWL-S [34] and WSMO [14].

OWL-S [34], which was previously named DAML-S [9], is an ontology for formally defining Web Services. An ontology-based description of Web Services has many advantages. First, it promotes the discovery of functionally-compatible components through the notion of a capability. In this sense, pioneering work by Paolucci *et al.* [44] defines an approach to assess functional compatibility between a provided service (advertisement) and a required service (request) by comparing the semantics of the inputs and outputs specified in their respective profiles: an advertisement matches with a request if every input in the request profile subsumes some input in the advertisement profile, and every output in the advertisement profile subsumes some output in the request profile. Second, it eases the construction of composition of services by making explicit the input, output, pre- and post-conditions of the services as well as their behaviours. Finally, and most importantly, it facilitates mediation by formalising both the meaning of the input/output and the behaviour of services. Vaculín *et al.* [52] define an approach for generating mediators between functionally-compatible client and service, both of which are modelled using OWL-S. First, they extract a set of representative executions of the client using its process specification. For each execution, they simulate the service process and use a planning algorithm in order to find the corresponding execution such that the client and the service can progress simultaneously. Then, for each pair of client and service executions, they use existing data mediators to perform the translations necessary to compensate for the differences between their input/output.

However, OWL-S only has had a qualified success because it specifies yet another model to define services. In addition, solutions based on process algebra and automata have proven more suitable for modelling and analysing the behaviour of components.

WSMO [14] is another ontology for modelling Semantic Web Services. WSMO considers mediators as first-class concepts and provides a runtime framework, the Web Service Execution Environment (WSMX), to specify, deploy, and execute mediators dynamically.

Semantic Mediation Bus. Instead of defining yet another ontology for Web Services, SA-WSDL [30] proposes a cost-effective solution to incorporate ontology reasoning in Web Services by augmenting service descriptions with annotations to: (i) define the semantics of operations and data by referring to concepts in a domain ontology, (ii) map the data syntax to the semantic definition of the

associated concept using XSLT[10], i.e., *lifting*, and (iii) derive the specific data structures from semantic concepts using XSLT also, i.e., *lowering*.

Even though SA-WSDL does not have the expressive power of OWL-S or WSMO as it represents neither the capability of services nor their behaviours, it is easier to integrate in existing systems including ESBs.

The Alion Semantic Mediation Bus [58] brings together SA-WSDL and ESB. While the ESB provides various plugins to support different middleware interaction protocols, services are described using SA-WSDL specifications, which enables the runtime translation of the actions of clients' and services' interfaces using the lifting and lowering functions. Nevertheless, as SA-WSDL does not support the modelling of behaviour, the Alion Semantic Mediation Bus focuses on action translations and does not coordinate the behaviours of clients and services. Moreover, as the capabilities are not represented either, the discovery of functionally-compatible clients and services cannot be achieved automatically.

Analysis. Semantic Web technologies, and ontologies in particular, enable the precise modelling of and reasoning about the meaning of the information exchanged between components. Semantic Web Services illustrate how ontologies can help to automate the discovery and composition of Web Services and facilitate mediation between them. However, mediation is often based on the definition of new ontologies and their use to infer the translations necessary to ensure the meaningful exchange of information between components. Furthermore, while modelling the behaviour of components is recognised as being essential, the logical theory behind ontologies is inappropriate for analysing components' behaviours. In addition, even though initial attempts to handle differences between components at the middleware layer are beginning to emerge through the concept of semantic mediation buses, they only deal with translations of actions and do not manage behavioural differences between components, at either the application or the middleware layers.

7 Mediator Synthesis as a Service

Over the years, mediation has been the subject of a great deal of work, both theoretical and practical. Table 1 summarises the solutions presented in previous sections. We can notice that although a lot of progress has been made, none of the proposed solutions is able to synthesise and deploy mediators that deal with both application and middleware differences and guarantee that the interaction between heterogeneous components is error-free.

- Software architecture solutions focus on reasoning about the composition of software components and define the requirements for mediation *but* do not specify how to synthesise mediators automatically.
- Middleware solutions facilitate the implementation of mediators *but* do not reconcile the differences between components at the application layer.

[10] Extensible Stylesheet Language Transformations – `http://www.w3.org/TR/xslt`

Table 1. Summary of mediation solutions

Pers.	Approach	The main idea	Evaluation
Software Architecture	Formal Reasoning about component interaction [20, 1]	Formal definition of component interaction to detect architectural mismatches	+ Formal basis for understanding mediation — No support for differences at the middleware layer — No automated generation of mediators
	Compositional approaches for connector development [49, 27]	Creating mediators from existing connector instances	
	Self-healing connectors [13, 17]	Recovery from component misuse by deploying connectors on the fly	
Middleware	Universal middleware [23, 53]	Provide an abstraction that masks the differences at the middleware layer	+ Support differences at the middleware layer — Developers need to specify or implement mediators at the application layer
	Middleware bridges [8, 7]	Direct translation between middleware messages	
	Service buses [40, 21]	Dealing with different middleware solutions via an intermediary infrastructure	
Formal Methods	Using a specification of the composed system [10, 5, 22]	Synthesise the mediator by selecting from all possible coordinations of the behaviours of components only those that satisfy the specification of the composed system	+ Automated analysis and coordination of components' behaviours + Guaranteed correctness of the interaction between components — Require a declarative specification of the correspondences between the actions of components' interfaces — No support for differences at the middleware layer
	Using a partial specification [31, 55, 43, 35]	Require the correspondences between actions to be available, and synthesise the mediator that guarantees that interaction between components is deadlock-free	
Semantic Web	Semantic Web Services [9, 14, 34]	Defining an ontology to support the inference of the necessary translations of the actions required by one component and provided by the other	+ Automated discovery of functionally-compatible components + Automated reasoning about the meaning of information — Partial support for behavioural differences — Partial support for middleware differences
	Semantic mediation bus [58]	Using semantic technologies within an ESB to automate message translation	

- Formal methods provide us with the foundations for coordinating the behaviours of components in order to guarantee the absence of errors in their interactions *but* assume that (i) the components use the same set of actions, (ii) a single specification of the composed system is given, or (iii) the correspondence between their actions is provided.
- Semantic Web solutions allow us to infer the translations necessary to ensure meaningful exchange of information between components *but* do not deal with the differences between components at the middleware layer.

Therefore, we propose to unify these different solutions by considering the various roles of mediators. Mediators act as (i) *translators* by ensuring the meaningful exchange of information between components, (ii) *controllers* by coordinating the behaviours of the components to ensure the absence of errors in their interaction, and (iii) *middleware* by enabling the interaction of components across the network so that each component receives the data it expects at the right moment and in the right format. More specifically, we can use ontology-based reasoning to automate the synthesis of translators, formal methods to synthesise controllers, and middleware solutions to realise and execute mediators. We can combine these solutions in a mix-and-match way to provide mediation as a service (see Fig. 3).

The first step consists in using domain knowledge to calculate the correspondences between the actions required by one component and those provided by the other, that is *translator synthesis* (see Fig. 3-❶). Indeed, a significant role of the mediator is to translate information available on one side and make it suitable and relevant to the other. This translation can only be carried out if there exists a semantic correspondence between the actions of the components, that is, *interface matching*. The main idea is to use domain-specific knowledge, described within an ontology for example, in order to select from sequences of actions of the components' interfaces only those which retain the meaning of the information exchanged and for which translations can automatically be computed. Interface matchings not only specify one-to-one correspondences between the actions of components but also many-to-many correspondences, which makes their computation very complex.

The second step is to explore the behaviours of the components in order to generate a process that ensures that whenever one of the components chooses a sequence of actions to execute, other components are ready to engage in a sequence of actions while there exist an interface matching relating these sequences of actions, that is *controller synthesis* (see Fig. 3-❷). The synthesised controller guarantees the correct interaction between the components by making them progress synchronously and reach a desirable state. Note that solutions such as MICS [3] tackles more than one role of mediators. MICS combines constraint programming and ontology reasoning to compute the correspondences between the actions used by the components, which are then used to synthesise a controller.

The last step entails the instantiation of the data structures expected by each component and their delivery according to the interaction pattern defined by the

middleware based on which the component is implemented, that is *middleware synthesis* (see Fig. 3-❸). Indeed, to enable the dynamic composition of highly-heterogeneous components, i.e. components featuring differences at both the application and middleware layers, a mediator must be synthesised which ensures that each component receives the data it expects at the right moment and in the right format.

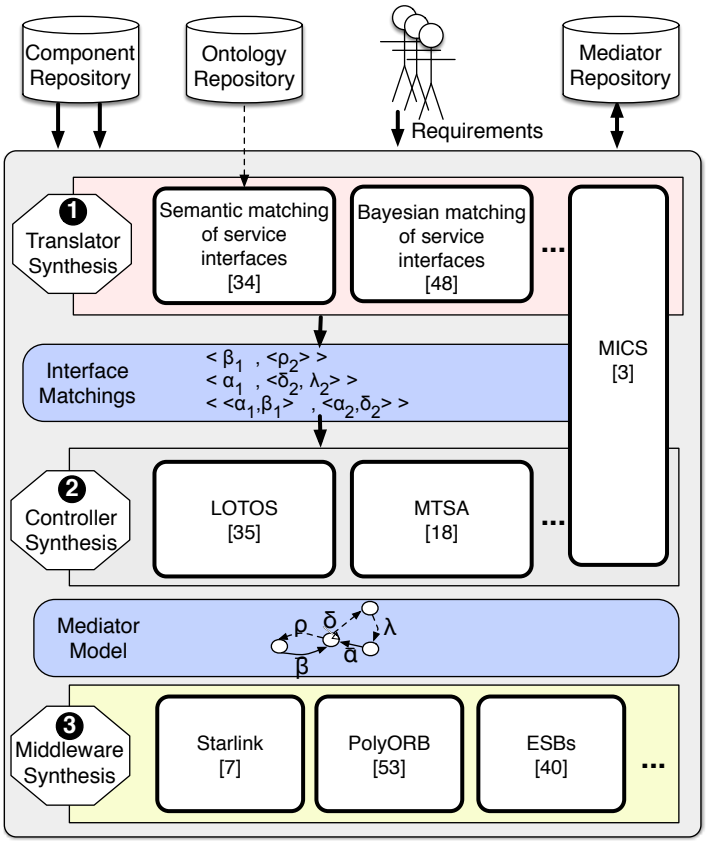

Fig. 3. Mediator Synthesis as a Service

8 Requirements and Mediation

Requirements and mediators may not seem to naturally fit together (see Fig. 4). On the one hand, requirements reside primarily in the problem space whereas mediators reside primarily in the solution space. That is, requirements reflect the

understanding of the environment, the need of stakeholders and the rationale behind the development of the proposed system. Mediator synthesis however focuses on the behaviour of individual components and how to enable them to interact with one another. Requirements are often refined by decomposing the problem into smaller ones whereas mediation aim to compose heterogeneous components to make a more complex behaviour emerge. On the other hand, the increasing deployment of mobile and ubiquitous computing technology makes the boundary between problem and solution worlds disappear. As a result, realising requirements through the collaboration of multiple existing components is more than simply desirable, it is fast becoming a necessity. But the remaining question is *how to bridge the gap between requirements and mediation?*

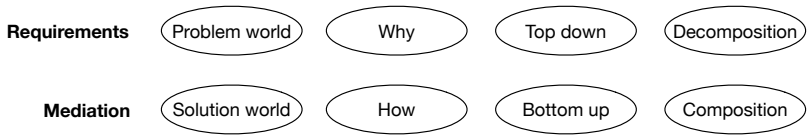

Fig. 4. Requirements vs mediation

Specifying requirements involves making explicit the environment properties under which these requirements must be satisfied [29]. More specifically, Jackson and Zave's framework for requirements engineering [29] makes explicit the relationship between requirements, specifications, and environment properties, which can be formalised as follows.

$$S, E \vdash R$$

where S denotes a system specification, E environment properties, and R requirements. Mediators are synthesised to realise a desirable property/requirement given a set of available components in a specific environment, which can be formalised as follows.

$$E, M \vdash R$$

where M denotes the synthesised mediator and we consider, without loss of generality, that the specifications of the available components are included in that of the environment.

When environment properties change (or the set of available components change) a new mediator must be synthesised to maintain the same requirement satisfied.

$$\text{Synthesise } M' \text{ such that } E', M' \vdash R$$

where E' denotes the updated environment properties and M' the new mediator.

However, it is not always possible to synthesise a mediator that will maintain the requirements satisfied whatever are the environment properties. D'Ippolito *et al.* [18] propose a multi-tier framework whereby a stack of mediators are synthesised to satisfy stronger requirements when making stronger assumptions about the environment. For example, a two level stack would be as follows.

$$\begin{array}{l}\text{Synthesise } M_1 \text{ such that } E_1, M_1 \vdash R_1,\\ \text{Synthesise } M_2 \text{ such that } E_2, M_2 \vdash R_2,\\ E_2 \text{ simulates } E_1, \text{ and}\\ M_2 \text{ simulates } M_1.\end{array}$$

where in the second tier, some strong assumptions about the environment are made E_2 and strong guarantees provided R_2 while weaker assumptions (E_2 simulates E_1) are made in the lower first tier but also weaker guarantees are provided. Nevertheless, this approach is unable to deal with unrelated environment properties or mediators. Consider for example the case where for the same environment properties, we can synthesise mediators to achieve only one requirement at a time:

$$\begin{array}{l}\exists M_1 \text{ such that } E, M_1 \vdash R_1,\\ \exists M_2 \text{ such that } E, M_2 \vdash R_2, \text{ and}\\ \nexists M \text{ such that } E, M \vdash R_1 \wedge R_2\end{array}$$

where R_1 and R_2 are two unrelated requirements. We must then decide which mediator to deploy, which necessitates explicit reasoning about requirements and their relationships.

Goal modelling frameworks such as KAOS [32] or i^* [56] are often used represent and reason about the relationships between multiple requirements as well as the associated domain properties. However, while goals, mainly expressed using Linear Temporal Logic (LTL) [45], have been extensively used in controller synthesis, it is not clear how goal modelling can be used for mediator synthesis. Cavallaro *et al.* [11] propose to extend the KAOS goal models in order to define a specifications of services, which are then instantiated at runtime. In this case, mediators are used to compensate for the differences between the discovered service instance and the service specification. Letier and Heaven [33] propose to use mediator (controller) synthesis to derive a machine specification that satisfies one goal under some domain specifications and then compose them to form a specification that satisfies a set of goals. Hence, combining requirement modelling and mediator synthesis help in dealing with multiple properties of the system composed of the multiple components and mediator.

Yet, rather than the synthesis of a machine specification, we may use requirements analysis to derive the appropriate specification and then implement this specification by using mediation to make multiple components collaborate. One concrete example is that of security. Determining the appropriate mechanisms that need to be deployed in order to protect assets from harm often requires trading off security against other requirements such as performance or usability and considering the value of the assets, and potential threats [26]. Adaptive security (sometimes called self-protection [57]) aims to enable systems to vary their protection in the face of changes in their operational environment. A requirements-driven approach for adaptive security enables the analysis and reasoning about the cost and benefit of the security controls. Salehie *et al.* [46] propose an approach in which a runtime model that combines goals, threats, and assets models is used to evaluate the cost and benefit of applying each security

control (i.e., the mechanism that needs to be deployed in order to protect assets from harm) and choosing the most appropriate one. Collaborative security [2] uses mediation to implement the appropriate security controls by composing components' capabilities at runtime.

9 Summary

Ask a software architect about mediation, and she will say that it is about the development of the software connector that enables components to interact successfully. Ask a middleware developer and she will tell you that it is about defining a connectivity infrastructure. Ask a formal methods expert and she will say that it is about computing a controller that enables the components to interact without errors. Ask a Semantic Web expert and she will tell you that it is about defining an ontology that enables reasoning about the meaning of the information exchange. In this chapter, we reviewed the literature on mediation from these four perspectives. We presented a multifaceted approach to mediation, which brings together the solutions of mediation from different perspectives. We also made a case for using requirements to help identify synergies and trade-offs between the many properties that a mediation solution need to deliver.

Acknowledgments.

We acknowledge SFI grant 10/CE/I1855 and ERC Advanced Grant no. 291652 (ASAP).

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions Software Engineering Methodology 6(3), 213–249 (1997)
2. Bennaceur, A., Bandara, A.K., Jackson, M., Liu, W., Montrieux, L., Tun, T.T., Yu, Y., Nuseibeh, B.: Requirements-driven mediation for collaborative security. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS. pp. 37–42 (2014)
3. Bennaceur, A., Issarny, V.: Automated synthesis of mediators to support component interoperability. IEEE Trans. Software Eng. 41(3), 221–240 (2015), `http://dx.doi.org/10.1109/TSE.2014.2364844`
4. Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific American 284(5), 28–37 (2001)
5. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174(3-4), 316–361 (2010)
6. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems, pp. 1–26. Springer Verlag (2011)

7. Bromberg, Y.D., Grace, P., Réveillère, L.: Starlink: Runtime interoperability between heterogeneous middleware protocols. In: International Conference on Distributed Computing Systems, ICDCS. pp. 446–455 (2011)
8. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Proc. of Middleware. pp. 21–41 (2009)
9. Burstein, M.H., Hobbs, J.R., Lassila, O., Martin, D.L., McDermott, D.V., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T.R., Sycara, K.P.: Daml-s: Web service description for the semantic web. In: International Semantic Web Conference, ISWC. pp. 348–363 (2002)
10. Calvert, K.L., Lam, S.S.: Deriving a protocol converter: A top-down method. In: Proc. of the Symposium on Communications Architectures & Protocols, SIGCOMM. pp. 247–258 (1989)
11. Cavallaro, L., Sawyer, P., Sykes, D., Bencomo, N., Issarny, V.: Satisfying requirements for pervasive service compositions. In: Proc. of the 7th Workshop on Models@run.time. pp. 17–22 (2012)
12. Ceriotti, M., Murphy, A.L., Picco, G.P.: Data sharing vs. message passing: synergy or incompatibility?: an implementation-driven case study. In: Proc. of the ACM Symposium on Applied Computing, SAC. pp. 100–107 (2008)
13. Chang, H., Mariani, L., Pezzè, M.: In-field healing of integration problems with COTS components. In: International Conference on Software Engineering, ICSE. pp. 166–176 (2009)
14. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Proc. of Business Process Management Workshop. pp. 130–143 (2005)
15. Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Computing Surveys 28(4), 626–643 (1996)
16. Coulouris, G.F., Dollimore, J., Kindberg, T., Blair, G.: Distributed systems: concepts and design, Fifth Edition. Addison-Wesley Longman (2012)
17. Denaro, G., Pezzè, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE. pp. 253–262 (2009)
18. D'Ippolito, N., Braberman, V.A., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proc. of the 36th International Conference on Software Engineering, ICSE. pp. 688–699 (2014)
19. Fielding, R.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California (2000)
20. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: Proc. of the 17th International Conference on Software Engineering, ICSE. pp. 179–185 (1995)
21. Georgantas, N., Issarny, V., Ben Mokhtar, S., Bromberg, Y.D., Bianco, S., Thomson, G., Raverdy, P.G., Urbieta, A., Cardoso, R.S.: Middleware architecture for ambient intelligence in the networked home. In: Nakashima, H., Aghajan, H., Augusto, J. (eds.) Handbook of Ambient Intelligence and Smart Environments, pp. 1139–1169. Springer (2010)
22. Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE Transactions on Services Computing 5(1), 72–85 (2012)
23. Grace, P., Blair, G.S., Samuel, S.: ReMMoC: A reflective middleware to support mobile client interoperability. In: Proc. of the OTM Confederated International Conferences CoopIS/DOA/ODBASE. pp. 1170–1187 (2003)

24. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5(2), 199–220 (Jun 1993), http://dx.doi.org/10.1006/knac.1993.1008
25. Gruber, T.: Ontology. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, pp. 1963–1965. Springer US (2009)
26. Haley, C.B., Laney, R.C., Moffett, J.D., Nuseibeh, B.: Security requirements engineering: A framework for representation and analysis. IEEE Trans. Software Eng. 34(1), 133–153 (2008)
27. Inverardi, P., Tivoli, M.: Automatic synthesis of modular connectors via composition of protocol mediation patterns. In: Proc. of the 35th International Conference on Software Engineering, ICSE. pp. 3–12 (2013)
28. Issarny, V., Caporuscio, M., Georgantas, N.: A perspective on the future of middleware-based software engineering. In: Proc. of the Workshop on the Future of Software Engineering, FOSE. pp. 244–258 (2007)
29. Jackson, M., Zave, P.: Deriving specifications from requirements: An example. In: Proc. of the 17th International Conference on Software Engineering, ICSE. pp. 15–24 (1995)
30. Kopecký, J., Vitvar, T., Bournez, C., Farrell, J.: SAWSDL: Semantic annotations for WSDL and XML schema. IEEE Internet Computing 11(6), 60–67 (2007)
31. Lam, S.S.: Protocol conversion. IEEE Transaction Software Engineering 14(3), 353–362 (1988)
32. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)
33. Letier, E., Heaven, W.: Requirements modelling by synthesis of deontic input-output automata. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 592–601 (2013), http://dl.acm.org/citation.cfm?id=2486866
34. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with OWL-S. In: Proc. of the World Wide Web conference, WWW'07. pp. 243–277 (2007)
35. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Transactions Software Engineering 38(4), 755–777 (2012)
36. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems 16(2), 46–53 (2001)
37. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004), http://doi.ieeecomputersociety.org/10.1109/MC.2004.48
38. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: The role of middleware in architecture-based software development. International Journal of Software Engineering and Knowledge Engineering 13(4), 367–393 (2003)
39. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proc. of International Conference on Software Engineering, ICSE (2000)
40. Menge, F.: Enterprise Service Bus. In: Proc. of the Free and open source Software conf. (2007)
41. Nezhad, H.R.M., Benatallah, B., Casati, F., Toumani, F.: Web services interoperability specifications. IEEE Computer 39(5), 24–32 (2006)
42. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. of the 16th International Conference on World Wide Web, WWW. pp. 993–1002 (2007)

43. Nezhad, H.R.M., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proc. of the 19th International Conference on World Wide Web, WWW (2010)
44. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Proc. of the First International Semantic Web Conference, ISWC (2002)
45. Pnueli, A.: The temporal logic of programs. In: Proc. of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57 (1977), http://dx.doi.org/10.1109/SFCS.1977.32
46. Salehie, M., Pasquale, L., Omoronyia, I., Ali, R., Nuseibeh, B.: Requirements-driven adaptive security: Protecting variable assets at runtime. In: Proc. of the 20th IEEE International Requirements Engineering Conference, RE. pp. 111–120 (2012)
47. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In: ICSE Workshop on Studies of Software Design. pp. 17–32 (1993)
48. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. Journal of Data Semantics IV pp. 146–171 (2005)
49. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proc. of the 25th International Conference on Software Engineering, ICSE. pp. 374–384 (2003)
50. Tanenbaum, A., Van Steen, M.: Distributed systems: principles and paradigms - Second Edition. Prentice Hall (2006)
51. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture: foundations, theory, and practice. Hoboken (N.J.) : Wiley (2009)
52. Vaculín, R., Neruda, R., Sycara, K.P.: The process mediation framework for semantic web services. International Journal of Agent-Oriented Software Engineering, IJAOSE 3(1), 27–58 (2009)
53. Vergnaud, T., Hugues, J., Pautet, L., Kordon, F.: PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In: Proc. of the 9th International Conference on Reliable Software Technologies Reliable Software Technologies, Ada-Europe. pp. 106–119 (2004)
54. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer 25(3), 38–49 (1992)
55. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and System, TOPLAS 19(2), 292–333 (1997)
56. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: Proc. of the 3rd IEEE International Symposium on Requirements Engineering, RE. pp. 226–235 (1997)
57. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. ACM Transactions on Autonomous and Adaptive Systems, TAAS (2014)
58. Zhu, W.: Semantic mediation bus: An ontology-based runtime infrastructure for service interoperability. In: Proc. of the 16th International on Enterprise Distributed Object Computing Conference Workshops, EDOCW. pp. 140 –145 (sept 2012)