A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet

Amel Bennaceur, Emil Andriescu, Roberto Speicys Cardoso, Valérie Issarny

▶ **To cite this version:**

Amel Bennaceur, Emil Andriescu, Roberto Speicys Cardoso, Valérie Issarny. A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet. Journal of Internet Services and Applications, 2015, pp.14. 10.1186/s13174-015-0027-3 . hal-01152426v2

HAL Id: hal-01152426

https://inria.hal.science/hal-01152426v2

Submitted on 26 May 2015

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

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

RESEARCH

A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet

Amel Bennaceur[1*], Emil Andriescu[2,3], Roberto Speicys Cardoso[3] and Valérie Issarny[2]

Abstract

Given the highly dynamic and extremely heterogeneous software systems composing the Future Internet, automatically achieving interoperability between software components —without modifying them— is more than simply desirable, it is quickly becoming a necessity. Although much work has been carried out on interoperability, existing solutions have not fully succeeded in keeping pace with the increasing complexity and heterogeneity of modern software, and meeting the demands of runtime support. On the one hand, solutions at the application layer synthesise intermediary entities, mediators, to compensate for the differences between the interfaces of components and coordinate their behaviours, while assuming the use of the same middleware solution. On the other hand, solutions at the middleware layer deal with interoperability across heterogeneous middleware technologies but do not reconcile the differences between components interfaces and behaviours at the application layer. In this paper we propose a unified approach for achieving interoperability between heterogeneous software components with compatible functionalities across the application and middleware layers. First, we provide a solution to automatically generate cross-layer parsers and composers that abstract network messages into a uniform representation independent of the middleware used. Second, these generated parsers and composers are integrated within a mediation framework to support the deployment of the mediators synthesised at the application layer. More specifically, the generated parser analyses the network messages received from one component and transforms them into a representation that can be understood by the application-level mediator. Then, the application-level mediator performs the necessary data conversion and behavioural coordination. Finally, the composer transforms the representation produced by the application-level mediator into network messages that can be sent to the other component. The resulting unified mediation framework reconciles the differences between software components from the application down to the middleware layers. We validate our approach through a case study in the area of conference management.

Keywords: Interoperability; cross-layer parsers and composers; mediation; dynamic composition; middleware

1 Introduction

Enabling the composition of functionally-compatible software components regardless of the technology they use and the protocols according to which they interact is a fundamental challenge in the Future Internet [1]. It has been the focus of extensive research, from approaches that identify the causes of interoperability issues and give guidelines on how to address them [2], to approaches that try to automate the application of such guidelines [3]. This challenge is exacerbated when heterogeneity spans the *application*, *middleware*, and *network* layers. At the application layer, components may exhibit disparate data types and operations, and may have distinct business logics. At the middleware layer, they may rely on different communication standards (e.g., SOAP and JMS) which define disparate data representation formats and induce different architectural constraints. Finally, at the network layer, data may be encapsulated differently according to the network technology in place. Heterogeneity at the network layer has partially been solved by convergence to a common standard (i.e., IP - Internet Protocol). In this paper, we focus on achieving interoperability across the application and middleware layers assuming the use of IP at the network layer.

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 spe-

*Correspondence: amel.bennaceur@open.co.uk
[1]The Open University, Milton Keynes, UK
Full list of author information is available at the end of the article
†Equal contributor

cific 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 deployed on Mac, Windows, and Linux machines can seamlessly access a SOAP-based Web Service deployed on a Windows server. However, a SOAP-based client cannot access a RESTful Web Service. 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. Middleware interoperability solutions [3] facilitate this task, either by providing an infrastructure to translate messages into a common intermediary protocol, as is the case for Enterprise Service Buses [4], or by proposing a Domain Specific Language (DSL) to describe the translation logic and to generate corresponding bridges [5]. These solutions, however, provide only an execution framework and still require developers to implement or specify the translations needed to enable the applications to interoperate.

Solutions oriented toward interoperability at the application layer, on the other hand, target higher automation and loose coupling. In particular, they rely on intermediary entities, *mediators* [6], to enforce interoperability by mapping the interfaces of functionally-compatible components and coordinating their behaviours. Solutions for the synthesis of mediators [7, 8, 9, 10, 11, 12, 13] focus on compensating for the differences between the components at the application layer, based on some domain knowledge, but without specifying how to deploy them on top of heterogeneous middleware solutions. As far as we know, only Starlink [14] allows binding application-layer mediators to different middleware solutions. However, Starlink requires the binding to be explicitly described in terms of the structure of messages that need to be sent or received by the components. Furthermore, this description is monolithic and binding cannot be reused across many applications.

In summary, existing solutions to interoperability have not fully succeeded in dealing with the increasing heterogeneity of components because of the following reasons: (i) they deal with middleware heterogeneity while assuming matching application components atop and rely on developers to provide all the translations that need to be made, (ii) they deal with behavioural mismatches at the application layer and generate corresponding mediators but fail to deploy them on top of heterogeneous middleware, or (iii) they deal with both middleware and application interoperability in conjunction but require the complete, and low-level specification of message structure using a general purpose DSL.

Furthermore, a critical issue for enabling heterogeneous components to interoperate is the ability to parse messages from the middleware layer into a format that can be handled by application-layer mediators and then concretise back (a.k.a. compose / unparse) the messages produced by application-layer mediators into middleware-layer messages. We refer to the process of parsing and composing a message as *message translation*. However, message translation is challenged by the encapsulation of data according to different middleware protocols, e.g., SOAP message encapsulated within HTTP for Web Services. As a result, implementing message translators requires dealing with multiple message formats and identifying the parts of the message corresponding to each protocol. What is needed is a declarative solution that facilitates the composition of multiple, and potentially heterogeneous, translators while taking into account the data dependencies between the application and middleware layers.

In this paper, we define a unified approach to deal with interoperability at both the application and middleware layers. We focus on client-service systems which are functionally-compatible, that is at some high level of abstraction the client requires a functionality that is provided by the service but is unable to interact successfully with it due to mismatching interfaces and behaviours. Our key contribution stems from the systematic and rigorous approach to generate complex message translators and their seamless integration with application-layer mediation techniques in order to manage cross-layer data dependencies. More specifically, we make the following contributions:

- *Composite Cross-Layer (CCL) message translators.* We devise an approach to automate the composition of message translators, called *CCL message translators*, that are able to process messages sent or received by software components implemented using different middleware solutions. We generate the message translators based on a declarative high-level specification that: (i) reuses implementations of message translators for legacy protocols (e.g., HTTP, SOAP, CORBA), (ii) easily integrates with interface-description and serialisation languages (e.g.,WSDL, XSD, ASN.1), and (iii) builds upon format-specific reverse-engineering tools (e.g., XML learning).
- *A unified mediation framework.* In previous work [13], we developed an approach based on ontology reasoning and constraint programming to

synthesise application-layer mediators automatically. We build upon this approach and extend it with CCL message translators to provide a unified mediation framework that deals with interoperability at both the application and middleware layers. This framework is capable of generating composite message translators as well as to synthesise application-layer mediators, which are deployed over a dedicated mediator engine.

- *Implementation and experimentation with a real-world scenario.* To validate our approach, we implemented a prototype tool and experimented it with heterogeneous conference management systems. Conference management systems provide various services such as ticketing, attendee management, and payment to organise events like conferences, seminars and concerts. Nevertheless, it is sometimes necessary to interact with different conference management systems. This is the case of Ambientic (`http://www.ambientic.com/en/`), which develops mobile software in the domain of Event Management (expos, trade shows, exhibitions, conferences). Depending on the event, organisers may choose to rely on different conference management systems. Our solution helps Ambientic integrate with different conference management systems transparently.

This paper is organised as follows. Section 2 introduces the interoperable conference management example, which we use throughout the paper to motivate and illustrate our mediation approach. Section 3 presents the proposed unified mediation framework that enables the generation of both CCL message translators and their integration with mediator synthesis at the application layer. Section 4 then surveys state of the art solutions to the generation and composition of message translators, thereby highlighting the variety of atomic message translators that need to be composed within CCL translators. The latter is the focus of Section 5, which details our approach to the generation of CCL translators by reusing and composing legacy ones. Section 6 describes a prototype implementation of the unified mediation framework and reports on the experiment we conducted using the interoperable conference management example. Finally, Section 7 concludes the paper and discusses future work.

2 The Interoperable Conference Management Example

To motivate and illustrate our approach, we consider the Ambientic application for event management, called U-Event (see Figure 1). U-Event embeds a *client component* implemented as an Amiando client (`http://developers.amiando.com/`). U-Event needs

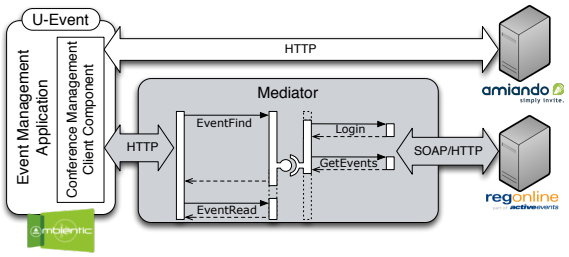


Figure 1: Interoperability scenario

to coordinate with a functionally-compatible service, Regonline (`http://developer.regonline.com/`). Instead of re-implementing the client component, the integration of the U-Event app with Regonline relies on our unified mediation framework.

In the following, we examine the challenges of enabling the Amiando client and the Regonline service to interoperate. The complete description of both systems is beyond the scope of this paper as they define more than 50 operations each. We thus concentrate on the following interaction: the client component must obtain a list of conferences based on keywords found in their title, and browse the information (such as dates or registration fees) of the obtained conferences. Amiando clients have to send an `EventFind` request containing the keywords to query. For security purposes, each Amiando client is assigned a unique and fixed `ApiKey` which must be included in every interaction with the service. The `EventFind` response includes a list of conference identifiers. To get some information about a conference, clients issue an `EventRead` request with the event identifier as a parameter. To produce the equivalent result, a Regonline client must first invoke the `Login` operation in order to obtain a session identifier `ApiToken`, which must be included in all subsequent requests. The Regonline client then sends a `GetEvents` request, which includes a `Filter` argument specifying the keywords to search for. The client gets in return the list of conferences matching the search criteria including their details. Both Amiando and Regonline are based on the request/response paradigm, i.e., the client issues a request which includes the appropriate parameters and the server returns the corresponding response. However, Amiando is developed according to the REST architectural style, uses HTTP as the underlying communication protocol, and relies on JSON (`http://www.json.org`) for data formatting. On the other hand, Regonline is implemented using SOAP, which implies using WSDL (`http://www.w3.org/TR/wsdl`) to describe the application interface, and is further bound to the HTTP protocol. Although the client component, which is an

Amiando client, requires some functionality provided by the Regonline service, it is unable to interact with it because of the mismatches described in the following.

Application-layer mismatches. To interoperate, components have to agree on the syntax and semantics of the operations they require and provide together with the associated input and output data. However, the same concepts (e.g., conferences, tickets, and attendees) may be expressed using different data types. To enable the components to interoperate, the data need to be converted in order to meet the expectations of each component. For example, to search for a conference with a title containing a given keyword, the Amiando client simply specifies the keyword in the title parameter, which is of type `String`. The Regonline `GetEvents` operation has a `Filter` argument used to specify the keywords to search for and which is also of type `String`. However, contrary to the WSDL description, the Regonline developer documentation specifies that this `String` field is in fact a C# expression and can contain some .NET framework method calls (such as `Title.contains(''keyword'')`), which is incompatible with the Amiando search string. The granularity and sequencing of operations is also very important. For example, the `GetEvents` operation of Regonline returns a list of conferences with the corresponding information. To get the same result in Amiando, two operations need to be performed.

Middleware-layer mismatches. Amiando is based on REST while Regonline is based on SOAP. Messages generated by both systems are incompatible and must be translated to allow them to interoperate. Moreover, the mechanisms provided by each middleware to describe the application interface are different: while SOAP-based Web Services rely on a standard interface description language (WSDL) to describe operations, there is no standard description language for RESTful services, although JSON is widely used, and in particular by Amiando.

Cross-layer data mismatches. Even though application and middleware layers are conceptually separate, in real-world scenarios the boundaries between them are ill-defined. This is due to multiple factors such as performance optimisation, simplified development or bad design decisions. For example, the `Login` operation of Regonline returns an `ApiToken` value, which is application-specific data. However, instead of including this token in subsequent operations at the application-layer encapsulation, it is inserted in the HTTP message-header (i.e., part of the middleware layer) as an optional field.

This example, although simple, demonstrates many problems that are faced by developers, and suggests why existing interoperability approaches still fall short in achieving interoperability. What is needed is a unified approach to interoperability that brings together and enhances the solutions that tackle interoperability at the application and middleware layers, and automates the generation of message translators and mediators.

3 A Unified Mediation Framework

We aim at providing a unified approach to support interoperability between functionally-compatible client-service systems by mediating their protocols from the application down to the middleware layers. Figure 2 depicts the main elements of our unified mediation framework where those with grey background are automatically synthesised. The framework revolves around two key elements: *CCL translator generator* and *synthesis of application-layer mediators*.

CCL translator generator: enables fast design of complex message translators while requiring minimal development effort by reusing existing implementations of atomic message translators, when available. Figure 2 depicts the main elements relating to *CCL translator generator*:

- *Atomic message translator*s transform one message format into an Abstract Syntax Tree (AST). An AST is a tree representation of the abstract syntactic structure of a protocol message. Each node of the tree denotes a data field of the message, and may contain metadata of the field. AST are a format commonly used in message translation and middleware technology. *Atomic message translator*s are reused and composed in order to generate *CCL message translators*.
- *Message Model* defines the strategy for assembling *Atomic message translators* in order to deal with the data encapsulation in different middleware solutions and cross-layer data dependencies. The message model also includes annotations that are integrated in the generated *Abstract Message Schemas*. Each rule or annotation in the *Message Model* is applied to an *Atomic message translator* at a particular node of its AST structure to solve or to annotate a cross-layer data dependency.
- *Abstract Message Schemas* is an abstract description of the component's interface that facilitates the synthesis of application-layer mediators. This schema composes and refines the AST schemas of a set of *Atomic message translator*s. Abstract messages of a generated *CCL message translator* validate the generated *Abstract Message Schema*.

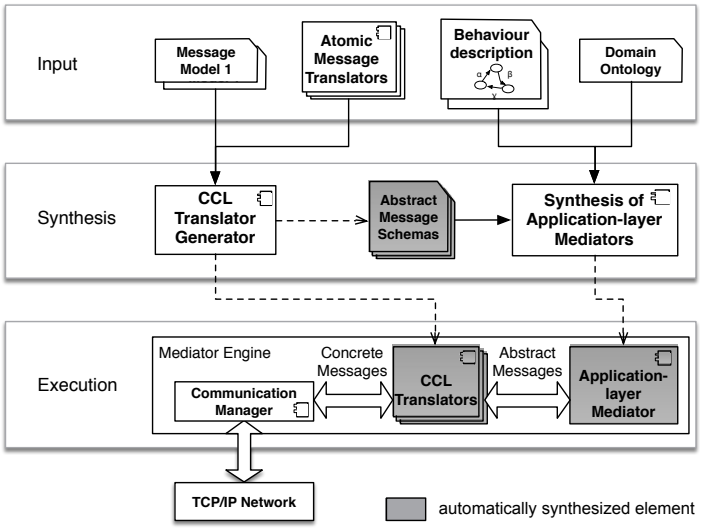


Figure 2: A unified mediation framework

Synthesis of application-layer mediators: is responsible for generating *application-layer mediators* based on the description of the interfaces and behaviours of the components involved, together with the associated *domain ontology*. The interfaces of the components are described using *Abstract Message Schemas*. The *behaviour* of a component then describes the ordering of the messages sent or received by this component in order to interact with other components in the environments. The behaviour of a component may be automatically extracted using automata learning techniques [15, 16, 17, 18].

The *synthesis of application-layer mediators* has been the subject of a lot of work, as surveyed in [19]. In their seminal paper, Yellin and Strom [7] propose an algorithm for the automated synthesis of mediators based on predefined correspondences between messages. By considering the semantics of actions, Vaculín *et al.* [20] are able to infer the correspondences between messages automatically. To generate the application-layer mediator, they generate all requester paths and find the appropriate mapping for each path by simulating the provider process. Cavallaro *et al.* [21] also consider the semantics of data and relies on model checking to identify mapping scripts between interaction protocols automatically. Nevertheless, they propose to perform the interface mapping beforehand so as to align the actions of both systems. However many mappings may exist and should be considered during the mediator generation. Indeed, the interface and behavioural descriptions are inter-related and should be considered in conjunction. Moreover, they focus on the mediation at the application layer assuming the use of Web services for the underlying middleware. Finally, Inverardi and Tivoli [11] propose 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.

The aforementioned research initiatives have made excellent contributions. However, in environments where there is little or no knowledge about the components that are going to meet and interact, the generation of suitable mediators must happen at runtime whereas in all these approaches, the mediator models or some mediation strategies and patterns are known a priori and applied at runtime. We have specifically developed a solution combining ontology reasoning and constraint programming to synthesise application-layer mediators automatically [13]. In the following, we briefly describe the gist of this approach while details can be found elsewhere [13]. Our main goal in this paper is to define a unified approach that deals with mediation at both the application and middleware layers rather then presenting the synthesis of application-layer mediators.

To synthesise an application-layer mediator, we begin by identifying the semantic correspondence between the messages sent by one component and those expected by the other component, that is interface matching. Indeed, a significant role of the mediator is to convert data available on one side and make it suitable and relevant to the other. This conversion can only be carried out if the data enclosed in the expected messages can be safely produced from the data embedded in the received messages. We formulate interface matching as a constraint satisfaction problem

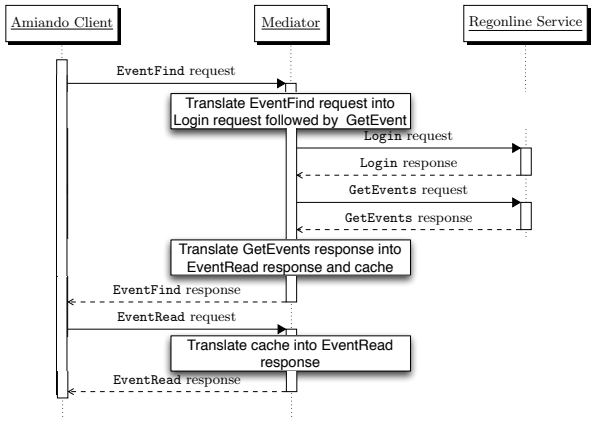


Figure 3: Illustrating the mediator between an Amiando client and the Regonline service

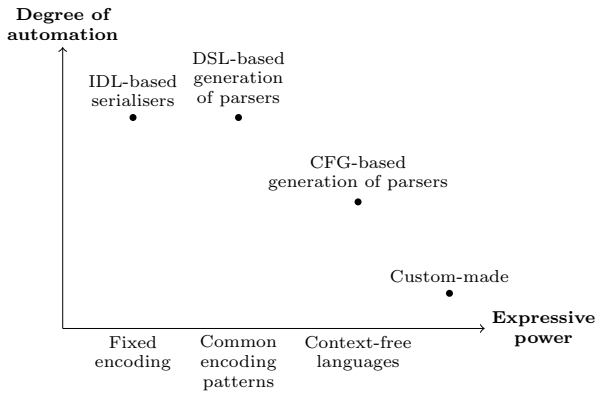


Figure 4: Parser design methods

and use constraint programming to solve it. We further incorporate the use of ontology reasoning within constraint solvers by defining an encoding of the ontology relations using arithmetic operators supported by widespread solvers. For each identified correspondence, we generate an associated matching process that performs the necessary data conversions between the actions of the components' interfaces. Then, we analyse the behaviours of components so as to generate the application-layer mediator that combines the matching processes in a way that guarantees that the components progress and reach their final states without errors.

Figure 3 illustrates the data conversion and behavioural coordination performed by the *application-layer mediator* that enables the *Amiando client* and the *Regonline service* to interoperate. The *application-layer mediator* intercepts the `EventFind` request sent by the *Amiando client* and transforms it into two invocations: `Login` and `GetEvent`. It generates the `EventFind` response based on the `GetEvents` response and is able to produce the responses of the following `EventFind` invocations. The reason is that the `GetEvents` includes a list of events while the `EventRead` requires only one event.

As depicted in Figure 2, the *Mediator Engine* enables the components to interoperate by executing the synthesised *application-layer mediator* while relying on the generated *CCL message translators* to deliver the messages in the expected formats. The *Communication Manager* keeps track of all network connections and pending message receptions. It support several IP transport protocols including TCP, UDP, TLS/SSL, and SOCKS.

4 Message Parsing and Composition: State of the Art and Analysis

In order to deal with cross-layer message translation and generate CCL message translators, it is essential to have a good knowledge and understanding of the various message syntaxes encountered in middleware technologies as well as the issues that can rise when composing heterogeneous message translators. In general, a message translator assures two functions: (i) parsing a stream of bits or characters, representing a network message in order to produce an AST, and (ii) processing an AST to produce a network message in the format expected by a given component. Most existing approaches focus on the parsing problem, which is, in the general case, the hardest. In this section, we present and analyse existing approches for generating and composing message translators.

4.1 Survey of Message Parsing Approaches

There exist a plethora of approaches to build message parser: some are optimised for low bandwidth overhead (e.g., Google's mechanism for serialising structured data known as Protocol Buffers), and others are specifically designed to facilitate interoperability (e.g., by using standard data serialisation formats). The forms in which parsers are available also differ: parsers can be precompiled components, or high-level descriptions in a domain-specific language. In Figure 4, we distinguish four major classes of approaches to build message parsers.

Custom-made parsers. These are parsers implemented in an *ad hoc* manner using a general purpose programming language. The composition of such parsers often requires adding "glue code" to adapt the data and interactions between the individual parsers.

CFG-based generation of parsers. An efficient alternative to implementing *custom-made* parsers is provided by parser generators (e.g., Yacc, Bison, ANTLR). Parser generators transform a user-provided Context-Free Grammar (CFG) into an executable component, which parses an input according to the specification given. While parser generators allow the extensible or even incremental [22] generation of parsers, they lack the ability of integrating and composing already existing parser implementations. The problem of parser composition in the context of *CFG-based generators* has already been addressed by Schwerdfeger *et al.* [23, 24] with a precise focus on *extensible programming languages*. Combinatory parsing [25] offers a set of primitive operations for modular parser composition. These operations can define parser composition with respect to the parser's input e.g., sequential composition, alternative parsing, optional parsing and repetition, or by applying a transformation to a parser's output (i.e., result-conversion). However, CFGs are a *non-compositional formalism* in the sense that compositions require in-depth modification of the base CFG derivation rules.

DSL-based generation of parsers. DSLs can be used by experts to specify parsers for complex message formats at a higher abstraction level, and in a more compact way, than CFGs. Solutions for generating parsers based on a DSL specification [26, 14, 27] focus on enabling interoperability of already existing systems. However, they are usually associated with a specific kind of message encoding pattern (e.g., text-based, XML, type-length-value encoding, etc.), and thus have a limited expressive power. Further, such approaches are not future proof as more message formats are expected to emerge, which will not be accounted for by DSLs that are defined according to known message encoding patterns. In addition, DSL-based solutions do not support composition and require messages to be defined in a monolithic way, which can easily become unmanageable for complex protocols.

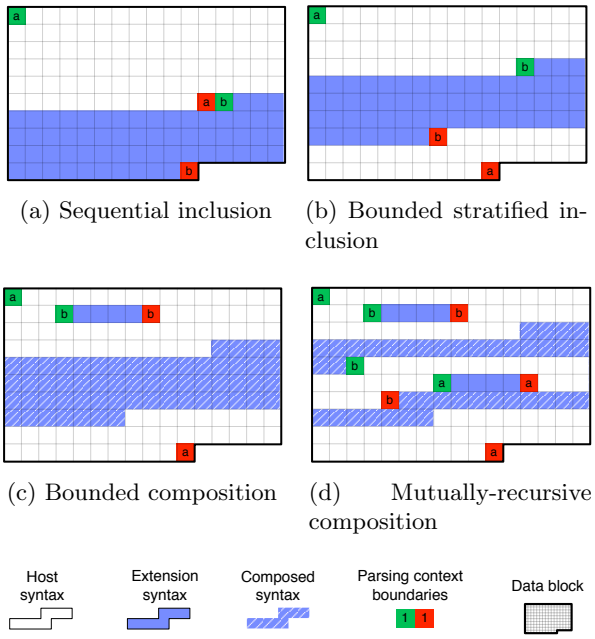
IDL-based serialisers. A different class of approaches for parser generation, use an Interface Description Language (IDL) that allows users to describe abstract structures of data using the IDL's type system. The description is passed to a compiler that generates source code, or compiled components capable of serialising & deserialising messages to & from the described data format. A major deficiency of IDL-based approaches (e.g., ASN.1, Protocol Buffers, CORBA OMG IDL, etc.) is that, while they can define an arbitrary data format, they usually support a fixed (or, in the case of ASN.1, a small set of) message encoding mechanism. Therefore, we view serialisers as a specific case of message parsers, with limited expressive power relative to the serialised message format (lower expressive power than DSL-based parsers). To facilitate the integration of serialisers with other systems, development environments, such as CORBA-based ones, provide an *IDL mapping* (`http://www.omg.org/spec/`) to data types (e.g., objects, lists, associative arrays, etc.) of various programming languages (e.g., Java, C++, Python, etc.). The mapping is supported by a separate IDL compiler for each programming language. Given the above limitation on message formats, the serialiser composition methods are also limited to IDL message-type encapsulation. While this method enables the reuse of IDL descriptions, it cannot be used to integrate other types of parsers.

All the aforementioned approaches are specific to a single *parsing method*. These are insufficient for the case of composite message parsing as, in real life situations, protocol stacks may use a mix of message formats that originate from *custom-made*, *CFG*, *DSL* and *IDL* generators. Hence, message parser composition must deal with the composition of heterogeneous message syntaxes and hence parsers.

4.2 Composing Heterogeneous Message Syntaxes

In a composite message format, ambiguity can occur between the encapsulating (outer) message format, called *host*, and encapsulatedmessage format, called *extension*. Parsing ambiguity is known to be a theoretically hard problem [28] but in communication protocols, several solutions are commonly implemented to deal with it:

- *Context-aware parsing* [29, 23] refers to methods and algorithms in which the scanner uses contextual information to disambiguate lexical syntax. This functionality allows a parser to carry out an alternative interpretation for the extension message. When they are ignored, the extension message can later be parsed by a second parser for that part of the message (e.g., CDATA escape sections in XML documents). The context change may be triggered by different mechanisms like *escape strings (or characters)*, or implicitly at predefined locations (e.g., SOAP envelope messages can only contain XML extensions, which may only be placed inside the `<head>` or `<body>` elements).
- *Lexical disambiguation.* Escape characters or character replacement can be used to resolve conflicts between the grammars of the host and extension. This method allows input lexemes (i.e., character sequences) from the base language to also appear in the extension language without causing ambiguities, which would otherwise result in parsing



(a) Sequential inclusion (b) Bounded stratified inclusion

(c) Bounded composition (d) Mutually-recursive composition

Figure 5: Illustrative example of the three classes of syntax composition

errors. For example, the string `Hello <World>` can be transformed into `Hello <World>` to disambiguate it from XML markup syntax.

- *Re-encoding.* Extension messages may also be entirely transformed into a different representation that does not conflict with the host syntax. This transformation can be done (i) by the host parser, in which case the behaviour is similar to escape sequences, (ii) by the extension parser, or (iii) by a separate component. For example Simple Mail Transfer Protocol (SMTP), which uses only text encoded messages, uses Base64 binary-to-text re-encoding to include binary data within SMTP messages.

In the following, we present the classes of syntax composition based on the principles of *context-aware parsing*. Figure 5 shows a schematic example for each class.

- *Sequential inclusion.* It is common in many protocols (e.g., protocols part of the TCP-IP stack, HTTP, etc.,) to compose messages by simply arranging the content in a sequential manner (e.g., one parser analyses a part of the input, and returns the remaining part in its result). In Figure 5a, we observe that the parsing context a a (corresponding to the host syntax) ends before the parsing context b b (corresponding to the extension syntax) begins.
- *Bounded stratified inclusion.* A middleware protocol parser is syntactically "unaware" of encapsulated messages, which are treated as a collection of binary data or arbitrary character strings. Because of this containment property, we can state that whenever two message parsers are composed to handle an encapsulated message format, they specialize (or restrict) the set of messages initially accepted. Thus, bounded stratified inclusion is a special case of syntax composition, which may only restrict the expressiveness of the base language, in the same sense explained by Cardelli *et al.* in [30]. Figure 5b illustrates such an example, where data associated with an extension syntax (shown in blue) is included at a specific point in the data of a message associated with a host syntax (shown in white). Although context b b is included in a a, they are properly delimited such that this message may be parsed even in the presence of lexical ambiguity between tokens of the host message and tokens of the encapsulated message.
- *Bounded composition* represents a generalisation of bounded stratified inclusion where the parsing context is not strictly delimitated. This means that lexemes from the host syntax can appear alongside lexemes of the extension syntax. Sections of the data block where this *composed syntax* is used (exemplified using hatched blue in Figure 5c) can be parsed neither by the host parser, nor by the extension parser. Unlike bounded stratified inclusion, bounded composition may include both *syntax extensions* and *syntax restrictions*. Syntax extensions expands the initial language with new message types, while syntax restrictions introduces intentional limitations on the expressiveness of a language.
- *Mutually-recursive syntax composition* refers to the case where the syntax of two distinct message formats can mutually be included inside one another. A technique commonly used to support this case of composition is *recursive descent parsing* (in particular implemented by parser combinators [25, 31]), where a composed parser is defined from a set of mutually recursive procedures. This class of syntax composition has been extensively studied in the domain of extensible programming languages [23, 24], where parser composition extends the syntax of a host programming language, for instance Java (e.g., context a a in Figure 5d), with another syntax, such as SQL (e.g., context b b). Intuitively, the syntax is mutually-recursive because SQL queries can appear within Java expressions, and, at the same

time, Java expressions can appear within SQL queries, allowing an unbounded chain of compositions. The same cannot be said about messages exchanged by protocol stacks where mutually-recursive compositions are unlikely given the fixed number of layers.

As far as we know, in existing protocol stacks, messages are encapsulated either using (a) sequential inclusion, or (b) bounded stratified inclusion. In [32], we further show that for these cases, heterogeneous parsers can be composed as black box functions (i.e., without requiring in-depth modification of the already existing parsers).

4.3 Atomic Message Translators

The *Atomic message translators* that can be used as input for composition are either *Legacy* (i.e., re-using an existing implementation) or *Generated* (i.e., generated at design-time).

*Legacy Atomic message translator*s are appropriate for middleware protocols given that they are based on industry-wide standards, with reference implementations widely available, and are unlikely to change frequently.

Generated Atomic message translators are useful for application-specific protocols, where changes in message structure are frequent. *Generated Atomic message translators* are further categorised depending on the availability of a message description language: *DSL* and *IDL-based* and Inferred. As the title suggests, some message formats can indeed be inferred automatically. This is the case when protocols represent/encode data using an extensible serialisation (e.g, JSON, YAML) or encoding format (e.g., ASN.1 –syntactical– - BER –lexical–, XSD –syntactical– - XML –lexical–)[1]. For this case to be applicable in a protocol mediation scenario, we obviously require a set of *Concrete Message Samples* that are used as input for type inference. In our experimental implementation, we rely on the tool Trang (`http://www.thaiopensource.com/relaxng/trang.html`) that can infer a schema from XML, JSON or other similar serialisation formats. Based on this schema, we automatically generate the corresponding *syntactical parsers*.

In the above, we make the assumption that parsers output ASTs using a uniform format that can be manipulated. In our implementation, we reduce the scope to object-oriented parser implementations. This is because AST instances represented as Objects may be examined or even manipulated at runtime using reflection and bytecode manipulation and may be easily serialised to other formats, like XML.

Assuming that all necessary *Atomic message translators* (either inferred, generated or off-the-shelf) for the mediated applications are available we generate a set of *CCL message translators* corresponding to the set of message types exchanged. In the *Amiando client* to *Regonline service* scenario, the set of *Atomic message translators* contains: *a*) legacy message translators for HTTP and SOAP, *b*) custom XML parsers generated from the WSDL/XSD description provided by Regonline and *c*) custom JSON [2] parsers for Amiando inferred using a set of pre-collected `Concrete Message Samples`.

5 Cross-Layer Mediation

In this section, we describe our approach for generating *CCL message translators* by composing multiple, and possible heterogeneous, *Atomic message translators*.

5.1 Composition of Message Translators

We mentioned that *Atomic message translators* are combined based on a *Message Model*. In Figure 6 we present a fragment of the *Message Model* describing the *Regonline service*. This description is used to generate the corresponding *CLL Translator* and Abstract Message Schema. A *Message Model* comprises three sections: `translator chaining`, `syntactic annotations`, and `semantic annotations`. The `translator chaining` section of the *Message Model* defines the composition of *Atomic message translators* to form the set of *CCL message translators* associated with an application. Each *CCL message translator* is generated according to an `operation` (i.e., a pair of request and response messages and associated data) of the component's interface. Using the `extension` composition rule, we declare how a specific field in the output (i.e., the output AST) of an *Atomic* or *CCL message translator* can be derived as input of a second Atomic message translator determined by the `identifier` tag. Generated Atomic message translator `extensions` require an extra `description` element containing a URI pointing to the message description and, optionally, a domain-specific `content` tag that specifies which part of the message description must be used, in the case where the provided description covers multiple operations. Field selection inside the AST is done using `path` expressions. For convenience, the syntax is borrowed from XPath (`http://www.w3.org/TR/xpath/`). Extension declarations may also

[1] Note the difference between: (i) *lexical parsers* that consume streams of characters or bytes and, in case of success, output a result in the form of an AST, and (ii) *syntactical parsers* that consume streams of tokens to produce the corresponding ASTs.

[2] Syntactical parsers defined on XML or JSON tokens.

```
<application name="Regonline">
  <operations>
    <operation>Login</operation>
    <operation>GetEvents</operation>
  </operations>
  <translator_chaining>
    <extension type="legacy" path="/" oper="*:Request">
      <identifier>mediation.http.HttpRequest</identifier>
    </extension>
    <extension type="legacy" path="/body">
      <identifier>mediation.soap.SoapMessage</identifier>
    </extension>
    <extension type="generated" path="/body/body" oper=
        "GetEvents:Request">
      <identifier>mediation.dynamic.wsdl.
          WsdlDefinedMessage</identifier>
      <description>https://www.regonline.com/api/default.
          asmx?wsdl</description>
      <content>GetEvents</content>
    </extension>
  </translator_chaining>
  <syntactic_annotation>
    <node path="/head/uri" oper="*:Request">
      <valuerestrict>
        <enumeration value="/api/default.asmx"/>
      </valuerestrict>
    </node>
    <node path="/head/headers[name=APIToken]/value"
        oper="GetEvents:Request">
      <extract fielddef="apiToken"/>
    </node>
    <node path="/head/soapAction" oper="GetEvents:
        Request">
      <map source="/body/body/soapAction"/>
    </node>
  </syntactic_annotation>
  <semantic_annotations>
    <node path="/head/apiToken" oper="GetEvents:
        Request">
      <!-- Domain knowledge -->
      <concept>SecurityToken</concept>
      <datascope>replay-only</datascope>
    </node>
  </semantic_annotations>
</application>
```

Figure 6: Fragment of the Message Model for the Regonline component

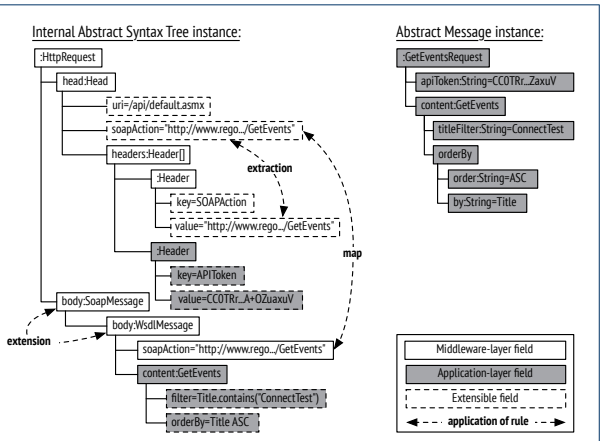


Figure 7: AST of Regonline GetEvents-Request

contain the optional attribute `oper`, which defines the operation for which the rule is relevant in the form `[Operation|*]:[Request|Response|*]`. Wildcards may be used on both sides of the attribute to specify that this rule applies to multiple operations or to both requests and responses.

We use the Message Model to create a tree structure based on the user defined `path` attributes and the ASTs corresponding to the referenced Atomic translators. We then recursively construct the composite message translators corresponding to each protocol `operation`, by applying composition and syntactical rules. This phase allows the composite message translators to produce *Internal AST instances*. As an illustration, a CCL AST instance of the Regonline `GetEvents Request` message is given in Figure 7. In this particular case, the initial input is parsed by an `HttpRequest` parser, then the `body` element encapsulating a SOAP message is further processed by a `SoapMessage` parser, and finally the SOAP `body` element is parsed by a dynamically generated `WsdlMessage` translator. The problem of inferring the data schema of the *Internal ASTs* is non-trivial. For this reason, in [32] we provide a formal mechanism, using tree automata, which based on a path expression (using a subset of the navigational core of the W3C XML query language XPath), generates an associated AST data-schema for the translator composition.

Secondly, we refine each *Internal AST* structure into a middleware-independent message-schema which defines the syntax of the *Abstract Message*. This process includes pruning all middleware-specific fields of the *Internal AST* schema, and also flattening the structure when possible without introducing ambiguity. The generated message schemas are enhanced with semantical annotations defined in the Message Model. This is the structure on which the application-layer mediator synthesis tool will reason, and infer appropriate mapping of data. Finally, we generate the functions necessary to transform *Abstract Messages* into their corresponding *Internal AST* representation, and the inverse.

5.2 Overcoming Cross-layer Data Dependency

We now take a closer look on how `syntactic` and `semantic` annotations can help solve cross-layer data dependencies and also support the synthesis of application-layer mediators.

A first step is assuring that all necessary data requirements are made explicit. While most abstract message structures (i.e., AST schemas) are automatically extracted from Atomic message translators and composed using our algorithm in [32], the `syntactic annotations` section of the Message Model further

augments this description. This may include specifying whether some fields are required or optional, or if there are any additional restrictions on the value of certain fields. For instance, in our scenario, the Regonline `GetEvents` operation accepts an optional `orderBy` parameter (see Figure 7) to specify the return order of conferences. If the application-layer mediator synthesis tool is unaware that this field is optional, it may fail to map an operation between components because a required input is not provided. Thus, we annotate this field as `optional`. For specific fields, the `valuerestrict` annotation allows specifying detailed value patterns for simple data types. While it may increase the complexity of the specification, this feature leads to a more precise data-mediation and message-validation than relying only on type-definition and/or semantical annotations.

Message formats may encapsulate sequences (e.g., lists or maps) of values of the same type. In some cases, the application may have requirements on the presence of a value, at a certain position. For example, the Regonline protocol requires that all requests except `Login` contain a session identifier provided as an HTTP header with the key `ApiToken`. The `extract` annotation allows making this requirement explicit with respect to the structure of the message by removing the specific field from the `headers` sequence, and reattaching it as a field at a higher level of the tree format.

When protocols are based on multiple middleware solutions, message composition may require data to be mapped internally across multiple translators. The `map` element enables to associate the values of middleware fields internal to a single CCL translator. For example, in the case of the message instance illustrated in Figure 7, the WSDL message translator field `body/body/soapAction` is mapped to the HTTP request header field `/head/soapAction`.

The last section of the Message Model, `semantic annotations`, enables the annotation of parsed data at various granularity. We support two types of semantic annotation: (i) domain knowledge (i.e., references to `concepts` in an ontology) and (ii) the scope of data. One may annotate an operation, a message, and/or any message field (either of complex or simple type). Such annotations support the synthesis of application-layer mediators in finding relevant matches between available data and data required to perform an operation. The `data scope` is important whenever applications configure the underlying middleware, causing application-specific data to be scattered over multiple layers. We mentioned that, in order to achieve mediation, we must identify and forward all application data. The element `datascope` set to `application` or `middleware` marks that the synthesis of application-layer mediators must consider this data as part of the application scope or, respectively, the middleware scope (in which case it should be ignored). However, the separation of middleware data is not sufficient as components may exhibit more complex data scoping. For example, Amiando uses a static key called `ApiKey` to control service access while Regonline uses a session id called `ApiToken`. Both data are instances of the same domain concept, but the mediator should never assign the `ApiToken` to `ApiKey` or vice-versa: Amiando will not recognise session keys created by Regonline and Regonline will not accept access keys generated by Amiando. Still, the application-layer mediator synthesis tool must map the `ApiToken` between the subsequent Regonline requests.

In response to the above data scoping challenge, we allow the `datascope` annotation to take the following values: (i) `middleware` when data is purely middleware specific and it should not be exposed to the application-layer mediator synthesis; (ii) `application` when data belongs to the application layer, and must be forwarded to the application-layer mediator; (iii) `replay-only` when application layer data should only be shared between the set of operations from the same component; (iv) `operation-only` when application layer data may only be included in certain operations; (v) `one-way` when application layer data may only flow in one direction, i.e., only Request or Response messages may include this data.

6 Implementation and Validation

We have implemented a prototype tool of the proposed mediation framework using Java, following the architecture described in Figure 8. The third-party tool and library dependencies for each component are mentioned between parentheses. The *Mediator Engine* implements the interfaces necessary to interact with the artefacts generated by the *Composite Message Translator Generator* and *Application-layer Mediator Synthesis (MICS)*.

In the case of the *CCL Message Translator Generator*, *Legacy* and *Generated* Atomic message translators are *chained*, *transformed* and *refined* using the bytecode manipulation library Javassist (`http://www.csg.is.titech.ac.jp/~chiba/javassist/`). To express richer constraints on the syntactic structure of ASTs beyond the basic means provided by Java Type definitions, we use the standard *Java Architecture for XML Binding*. In this way, each class structure is bound to an XSD schema. Since value-restrictions, as described by the *Message Model*, cannot be injected as compile-time JAXB annotations, they are transformed to a *JAXB External Binding Customization File*. Generated Atomic message translator are

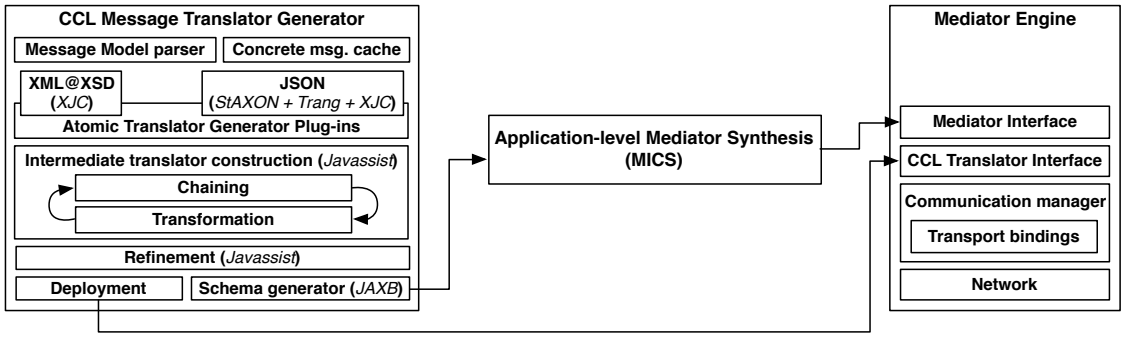


Figure 8: Implementation of the unified mediation framework

obtained using external tools, which are integrated as plug-ins. XJC (http://jaxb.java.net) is used for generating XML message translators based on XML Schemas. Since there is no well-established data-schema for JSON we use StAXON (https://github.com/beckchr/staxon/) tool to transform JSON messages to XML before learning their data-schema using the XML learning tool *Trang*. We consider the integration of additional Atomic message translator generators like, for example, *Java Asn.1 Compiler* (http://sourceforge.net/projects/jac-asn1/) for *ASN.1* parser specifications.

In what follows, we assess our approach by comparing the time to perform a conversation in the mediated and non-mediated case between Amiando client/service and Regonline client/service. Figure 1) shows the result. On the server-side, we use the services operated by Amiando and Regonline. On the client-side we use a Java implementation provided by Amiando, while for Regonline, we partially generate the client source-code using the provided WSDL service description.

We first specify a Message Model (https://www.rocq.inria.fr/arles/software/ccl-Mediation-Framework/) for each system, as well as two message samples containing the JSON formatted responses of the Amiando service. The composite message translator Generator is then able to generate eight different composite message translators (listed in Figure 10) and their associated *Semantically Annotated XSDs* (http://www.w3.org/2002/ws/sawsdl/). The SAXSDs are obtained by injecting semantical annotations obtained from the Message Models into the XSD schemas generated using JAXB. At runtime, MICS generates the two mediators given the SAXSDs, a conference management ontology and the behavioural descriptions of each system.

We compare the mediated execution-time with the non-mediated case. Each test was repeated 30 times,

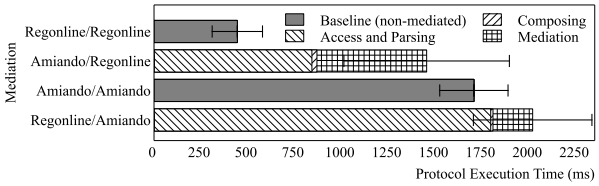


Figure 9: Comparison between mediated and non-mediated executions

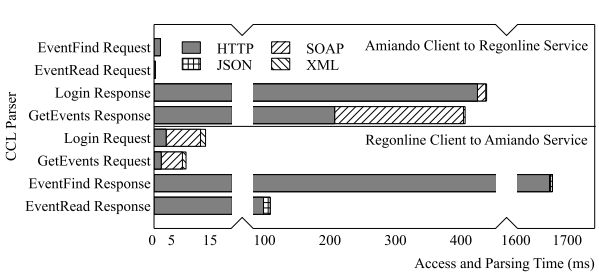


Figure 10: Access & Parsing time decomposed by Atomic parsers

in similar conditions, and connection delays were excluded (e.g., opening sockets, SSL handshake, etc).

In Figure 9, we evaluate the execution-time overhead of the mediation. Since this test is performed using the real online services, the response time varies depending on the network conditions. As expected, the mediated execution-time is superior to the non-mediated case, given that the number of messages exchanged is doubled. We show the decomposition of the execution-time for mediation, composing and access/parsing. Network access and parsing cannot be distinguished in this case because parsing is done in multiple steps when data is available on the communication channel. While the overhead of mediation and message composition is low, we see that parsing and network reception introduce

the largest overhead. This is why, in Figure 10, we detail the decomposition of parsing time over each Atomic parser used internally by a specific generated translator. We see that the `EventFind` response message parsing has a peak of 1662 ms. We also observe that the entire time is associated with the HTTP parser, and given that the size of the message is only 869 bytes, we can conclude it is almost entirely due to the response delay of the Amiando Service. The same reasoning applies for the `GetEvents` response message of the Regonline service, but in this case 197 ms are associated with the SOAP parser which is chained to parse the HTTP response's payload (the HTTP body). Knowing that in this particular implementation, the SOAP parser does not wait for network access, we observe that the SOAP Atomic parser introduces an important SOAP-Envelope parsing overhead. This observation confirms that the Amiando/Regonline (i.e., Amiando Client mediated to the Regonline Service) mediator execution-time (in Figure 9) can be reduced by using a more efficient SOAP Atomic parser. Hence, we can conclude that our mediation approach introduces an acceptable overhead while enabling seamless interoperability between two originally incompatible systems.

7 Conclusion and Future Work

Interoperability is a very challenging topic. Over the years, interoperability has been the subject of a great deal of work, both theoretical and practical. However, existing approaches focus on achieving interoperability either at the application or middleware layer. This paper presented a unified mediation framework to achieve interoperability from application down to middleware layers. We have shown via our implemented prototype that the framework successfully enables interoperability in a transparent way, while introducing acceptable overhead.

Future work includes increasing automation by inferring, at least partially, the Message Model by cooperating with discovery mechanisms and packet inspection software. We also intend to experiment with various learning techniques, both active and passive, for the inference of component behaviour. Finally, incremental re-synthesis of mediators and, runtime refinement of composite message translators would be useful in order to respond to changes in the individual systems or in the ontology. A further direction is to consider improved modelling capabilities that take into account the probabilistic nature of systems and the uncertainties in the ontology. This would facilitate the construction of mediators where we have only partial knowledge about the system.

Competing Interests

The authors declare that they have no competing interests.

Author details

[1]The Open University, Milton Keynes, UK. [2] Inria, Paris-Rocquencourt, France. [3] Ambientic, Paris, France.

References

1. Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadis, P., Autili, M., Gerosa, M.A., Hamida, A.B.: Service-oriented middleware for the future internet: state of the art and research directions. J. Internet Services and Applications **2**(1), 23–45 (2011). doi:10.1007/s13174-011-0021-3
2. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: ICSE (1995)
3. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: LNCS SFM-11 (2011)
4. Chappell, D.A.: Enterprise Service Bus. O'reilly Media, ??? (2004)
5. D.Bromberg, Y., Grace, P., Réveillère, L.: Starlink: Runtime Interoperability between Heterogeneous Middleware Protocols. In: ICDCS (2011)
6. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer **25**(3) (1992)
7. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM TOPLAS **19**(2) (1997)
8. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems **16**(2), 46–53 (2001)
9. Spitznagel, B., Garlan, .: A compositional formalization of connector wrappers. In: ICSE (2003)
10. 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)
11. 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)
12. 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)
13. Bennaceur, A., Issarny, V.: Automated synthesis of mediators to support component interoperability. IEEE Transactions on Software Engineering **41**(3), 221–240 (2015)
14. Bromberg, Y.D., Grace, P., Réveillère, L., Blair, G.S.: Bridging the Interoperability Gap: Overcoming Combined Application and Middleware Heterogeneity. In: Middleware (2011)
15. Oldham, N., Thomas, C., Sheth, A.P., Verma, K.: METEOR-S web service annotation framework with machine learning classification. In: SWSWPC, pp. 137–146 (2004)
16. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. of the International Conference on Software Engineering, ICSE, pp. 501–510 (2008)
17. Krka, I., Brun, Y., Popescu, D., Garcia, J., Medvidovic, N.: Using dynamic execution traces and program invariants to enhance behavioral model inference. In: Proc. of the 32nd International Conference on Software Engineering, ICSE (2), pp. 179–182 (2010)
18. Bennaceur, A., Issarny, V., Sykes, D., Howar, F., Isberner, M., Steffen, B., Johansson, R., Moschitti, A.: Machine learning for emergent middleware. In: Proc. of the Joint Workshop on Intelligent Methods for Software System Engineering, JIMSE (2012)
19. Issarny, V., Bennaceur, A.: Composing distributed systems: Overcoming the interoperability challenge. In: Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures, pp. 168–196 (2012)
20. 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)
21. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: ICSOC (2009)

22. Heering, J., Klint, P., Rekers, J.: Incremental Generation of Parsers. In: PLDI (1989). doi:10.1145/73141.74834. http://doi.acm.org/10.1145/73141.74834
23. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable Composition of Deterministic Grammars. PLDI (2009). doi:10.1145/1543135.1542499
24. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable Parse Table Composition for Deterministic Parsing. In: SLE (2010)
25. Swierstra, S.D.: Combinator parsers - from toys to tools. ENTCS **41** (2000)
26. Burgy, L., Reveillere, L., Lawall, J., Muller, G.: Zebu: A Language-Based Approach for Network Protocol Message Processing. IEEE TSE (2011). doi:10.1109/TSE.2010.64
27. Borisov, N., Brumley, D.J., Wang, H.J.: A Generic Application-Level Protocol Analyzer and its Language. In: NDSS (2007)
28. Basten, H.: Ambiguity detection methods for context-free grammars. Master's thesis, Universiteit van Amsterdam (2007)
29. Van Wyk, E.R., Schwerdfeger, A.C.: Context-aware Scanning for Parsing Extensible Languages. In: GPCE (2007). doi:10.1145/1289971.1289983. http://doi.acm.org/10.1145/1289971.1289983
30. Cardelli, L., Matthes, F., Abadi, M.: Extensible grammars for language specialization. In: DBPL (1994). http://dl.acm.org/citation.cfm?id=648290.754352
31. Moors, A., Piessens, F., Odersky, M.: Parser combinators in scala. KU Leuven, CW Reports vol:CW491. (2008)
32. Andriescu, E.-M., Martinez, T., Issarny, V.: Composing Message Translators and Inferring their Data Types using Tree Automata. In: Proc. of the 18th International Conference on Fundamental Approaches to Software Engineering, FASE, London, United Kingdom (2015). https://hal.inria.fr/hal-01097389