Dynamic Data Adaptation for the Synthesis and Deployment of Protocol Mediators

Emil-Mircea Andriescu

Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Inria Paris-Rocquencourt / Equipe-projet MiMove

Adaptation dynamique de données pour la synthèse et le déploiement de protocoles de médiation

Par **Emil-Mircea Andriescu**

Thèse de doctorat en Informatique

Dirigée par Valérie Issarny

Présentée et soutenue publiquement le 08 02 2016

Devant un jury composé de :

Khalil Drira (CNRS, FR)	Rapporteur
Massimo Tivoli (Université de L'Aquila, IT)	Rapporteur
Florent Jacquemard (IRCAM et Inria, FR)	Examinateur
Pierre Sens (UPMC, FR)	Examinateur
Pierre-Guillaume Raverdy (Ambientic, FR)	Examinateur
Valérie Issarny (Inria, FR)	Directrice de thèse

Pierre and Marie Curie University

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Inria Paris-Rocquencourt / MiMove Project-Team

Dynamic Data Adaptation for the Synthesis and Deployment of Protocol Mediators

By **Emil-Mircea Andriescu**

Doctoral thesis in Computer Science

Under the supervision of Valérie Issarny

Presented and defended publicly on 08 02 2016

In front of a jury composed of:

Khalil Drira (CNRS, FR)	Reviewer
Massimo Tivoli (University of L'Aquila, IT)	Reviewer
Florent Jacquemard (IRCAM and Inria, FR)	Examiner
Pierre Sens (UPMC, FR)	Examiner
Pierre-Guillaume Raverdy (Ambientic, FR)	Examiner
Valérie Issarny (Inria, FR)	Advisor

"The world is marching towards fragmented islands of communication connected via fragile pathways."

Vidya Narayanan[1]

[1]Vidya Narayanan is the former co-chair of the Network-based Mobility Management (NETLMM) working group at IETF from 2003 to 2010.

Abstract

In most systems available today interoperability is provided as a static capability that is the result of manually designed and hand coded integration. In consequence, a substantial number of functionally-compatible systems are not conceived to be interoperable.

The focus of this thesis is to enable automated protocol interoperability for systems, services and applications through the means of dynamically synthesised protocol mediators. Protocol mediators represent concrete software components which can coordinate interactions between two or more functionally-compatible systems, relying on various means of communication (IP networks, personal area networks, inter-process communication, shared memory, etc.). Dynamically synthesised mediators should allow applications to seamlessly adapt to a priori unknown protocols, support the evolution of such protocols while circumventing real-world system constraints, such as those introduced by device mobility and operating system differences.

In this thesis we focus on the research problems related to automating the process of data adaptation in the context of protocol mediation. Data adaptation is a key phase in protocol mediation that cannot be solved independently. This strong dependence becomes visible when systems relying on multilayer protocol stacks have to be made interoperable, despite cross-layer dependencies inside the exchanged data. There is the need of a framework that synthesises mediators while taking into account cross-layer data adaptation.

Key Words

Interoperability, Protocol Mediation, Data Adaptation, Protocol Stacks, Message Formats, Message Translation, Type Inference

Table of contents

Chapter 1

Introduction

Contents

Interoperability, or the lack of, is probably best understood in the domain of telecommunications. Take for instance the implications of interoperability on public safety. In the federal 9/11 Commission Report [1] regarding the attacks on the Pentagon and World Trade Center, the commissioners noted that "the capabilities of communications systems lacked the ability to communicate across department lines", meaning that police units could not communicate with fire units, or with ambulance units on site over radio. This led to unprecedented amount of money being spent (estimated to range up to five billion dollars) on radio equipment and infrastructures.

Just like communications equipment, there are many other systems that sustain the backbone of modern society, and although they serve for common goals they are still unable to interoperate. Another example can be medical devices that allow, life support, automatic decision/diagnostic support and medication checking in real-time. These systems have considerable communication capabilities that allow them to interact with each others and entities around them. However, just like the above, such systems do not cross "department lines". By this we mean that, either equipment from different vendors cannot communicate and work together directly, or that integration with with larger systems, such as electronic health records (EHRs) is not facilitated.

We posit that interoperability should be enabled between all compatible systems, starting from the most insignificant to the ones of critical importance for society. Yet, some people might regard this "proclamation" of universal interoperability with skepticism. Indeed, some electronic devices are known to be "self-sufficient", meaning that they do not require sharing data and functionalities with other systems, or the ability to be integrated as a component of a larger system. However, this view is rapidly changing with the emergence of the Internet of Everything (IoE), one of the hottest topics of today. For instance, it is no longer a thing of science fiction for a toothbrush to record your brushing activity as data charts that you can display on your mobile phone and share it with dental professionals (`http://connectedtoothbrush.com/`). And this revolution is not limited to household goods. It is fast becoming an accepted fact that modern systems are productive and cost-effective only if they can interoperate with other systems, sharing with them data and functionalities [2].

This leads us to give the definition of system interoperability. In what follows, we adhere to Tanenbaum's definition:

Definition 1 *(Interoperability) "Interoperability characterises the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard." [3]*

Interoperability is provided, in most cases, as a static capability which is the result of manually designed and implemented integration. Because of this, a substantial number of *functionally-compatible* systems are not conceived to be interoperable, where we define functional compatibility as:

Definition 2 *(Functionally-compatible systems) Two or more systems are said to be functionally-compatible if they require / provide at least one coarse-grained functionality (e.g., voice communication, vehicle control, video streaming, etc.) which is conceptually equivalent, but not necessarily having a technically compatible implementation. The two systems must have compatible roles, in the sense that one provides a functionality that the other requires (e.g., one system is a Web server while the other is a Web browser.).*

1.1 Protocol interoperability

While hardware interoperability is critical today, it is becoming less and less relevant as systems are becoming more complex, opening the door to a rather newer issue, that is *software interoperability*. Any interaction between interconnected software systems is assured today by either (i) the exchange of files via a common set of data formats or (ii)

through the use of compatible *communications protocols*. In this thesis we focus on the latter. By extension of the Definition 2 above, the interoperability of *communications protocols* is necessary whenever two or more systems that serve for a common purpose (e.g., file sharing, journey planning, conference management, etc.) were implemented to communicate using different communication protocols.

While the definition of a protocol is common knowledge, for the sake of rigour, we include the following definition to which we adhere:

> **Definition 3** *(Protocol) "A set of rules or procedures for transmitting data between electronic devices, such as computers. In order for computers to exchange information, there must be a preexisting agreement as to how the information will be structured and how each side will send and receive it." (Encyclopaedia Britannica)*

Simply put, a communications protocol defines the rules for sending data from one system in a network to another system. These data are encapsulated in messages. Typically, a protocol defines the following aspects of messages: (i) format (i.e., how messages are formed) (ii) synchronisation and direction (i.e., in which order they can be either sent or received). However, more complex protocols, such as the class of streaming protocols, define other mechanisms like: congestion control, loss control, error detection and routing.

Protocol interoperability is a vast research domain and proposed solutions to enable cooperation between functionally-compatible systems focus on precise aspects of the problem. Enabling interoperability of functionally-compatible software components regardless of the technology they use and the protocols according to which they interact is a fundamental challenge in Software Engineering [4]. 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 [5], to approaches that try to automate the application of such guidelines [6]. We group interoperability solutions as shown in Table 1.1.

From protocol layers to protocol stacks: In order to account for the complexity of modern systems (and systems of systems), protocols are rarely used independently but rather as a composition of layers, where each layer addresses (or abstracts) a specific aspect of the interaction. As we illustrate in Figure 1.1, systems commonly use protocol stacks composed of the following layers:

1. *TCP/IP network layers*

   (a) *Physical network:* Defines the characteristics of the network hardware. Common protocols include: Ethernet (IEEE 802.3), Token Ring, RS-232, FDDI.

Interoperability by conformance	We refer to methods of explicitly designing systems to be interoperable by conforming to a specific standard or methodology. Such methods closely follow the Definition 1 on interoperability.
Protocol conversion	Solutions that allow incompatible components to be made interoperable by specially designed software adapters. The adaptation is done at a low abstraction level, and thus, it is highly coupled with the components' implementation.
Interoperability platforms	Solutions which attempt to solve interoperability by integrating legacy components by alignment to a common framework, middleware or service-bus.
Dynamic protocol mediation	Approaches enabling incompatible systems to interact through the means of intermediary software components, called mediators, which may be generated automatically, to a certain degree. Mediation is done at a higher abstraction level than protocol conversion and it should generally be independent to the component implementation.

Table 1.1 – Common solutions towards protocol interoperability.

(b) *Data link:* Handles the transfer of data across the network media. Common protocols include: PPP, IEEE 802.2

(c) *Internet:* Manages data addressing and delivery between networks. Common protocols include: IPv4, IPv6, ARP, ICMP

(d) *Transport:* Manages the transfer of data and validates received data integrity. Common protocols include: TCP, UDP, SCTP

2. *Middleware layers:* Middleware is a software that allows other software to interact by providing a set of homogeneous communication primitives and procedures. Here we refer to Middleware that are used as a protocol layer between the TCP/IP transport layer and the application layer or yet another middleware layer. Specifically we refer to Message Oriented Middleware (MOM) and RPC Middleware. Common protocols include: SOAP, CORBA/GIOP, Java RMI, HTTP.

3. *Application layer.* It is the top-most layer of a protocol stack. All the preceding layers should be seen as components that cooperate to support the application layer. The application protocol allows an application or a service to perform a set of tasks for a user. The order and parameters of such tasks are controlled directly by the user using an interface (graphical or text based), or indirectly in the case of services. Here we refer to both standard protocols and proprietary application protocols.

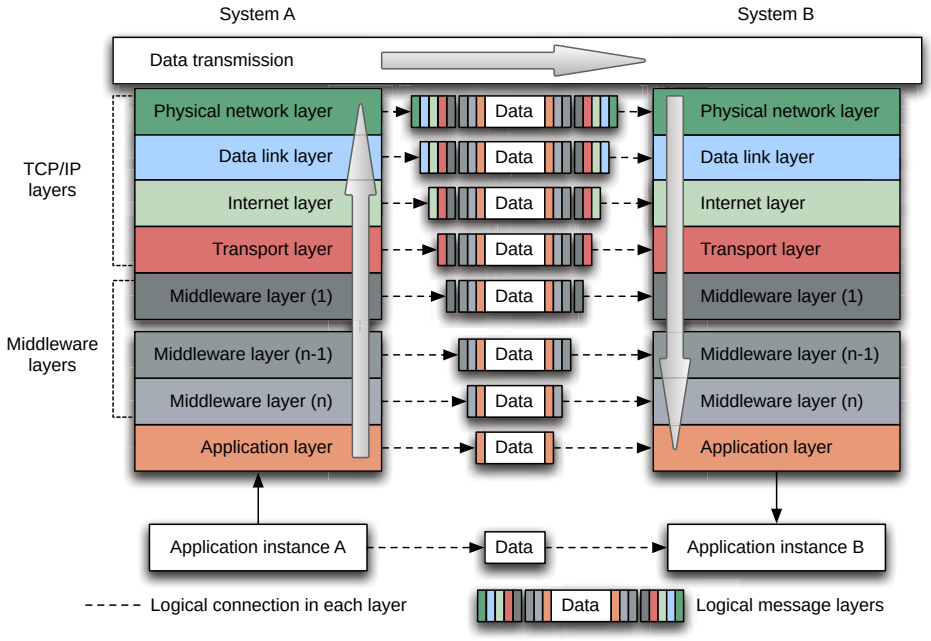


Figure 1.1 – Common protocol layers used by applications.

1.2 Data adaptation

While this decomposition of protocols into layers is beneficial for abstracting the design of complex systems it also represents a major challenge in protocol interoperability. This is because layers can rarely be fully decoupled due to performance optimisations and other design rationale. For instance, *logical message layers*, as they were illustrated in Figure 1.1, differ substantially from concrete message layers in the sense that data specific to one logical layer might be encapsulated inside the message part corresponding to a different layer.

In this thesis we focus on a specific problem in the context of protocol interoperability, that is *data adaptation*. Data adaptation refers to the following problems:

1. *Mapping between concrete message formats.* When two functionally-compatible protocols must be made interoperable, messages from one protocol must be translated into a format supported by a second protocol and the vice versa. This process involves mapping data values from one format towards another message format.

2. *Translation between concrete message formats and common intermediary message formats.* Direct mapping between heterogeneous message formats is an inefficient approach because a different mapping is required for each pair of message formats.

For this reason, most existing approaches towards data adaptation choose to transform messages into a common intermediary message format prior to realising the mapping of data.

3. *Mapping between common intermediary message formats.* Even when the message formats are homogeneous, the mapping of data is rarely trivial because data semantics and data structures (i.e., data models) differ from application to application and from middleware to middleware. For this reason, it is often required to analyse and devise more advanced methods for data mapping.

Our thesis statement is the following:

> "Functionally-compatible systems should be able to interoperate. Dynamically synthesised mediators allow applications to adapt to a priori unknown protocols, support the evolution of such protocols while circumventing real-world system constraints, such as those introduced by device mobility and operating system differences. Data adaptation is a key phase in protocol mediation that cannot be solved independently. This strong dependence becomes visible when systems relying on multilayer protocol stacks have to be made interoperable, despite cross-layer dependencies inside the exchanged data. There is the need of a framework that synthesises mediators while taking into account dynamic data adaptation. "

As we mention in our thesis statement, even though we focus on data adaptation, the research we present in this work spans beyond this aspect. This is because data adaptation cannot be solved independently from other phases of designing protocol interoperability solutions. Furthermore, in order to assess the effectiveness and relevance of our contributions with concrete systems and applications, we were required to design and implement complete interoperability solutions.

1.3 An overview of this work

In the previous section, we introduced the context of protocol interoperability and we underlined the problems concerning data adaptation. A more detailed description including state of the art approaches for *protocol interoperability*, in general, and *data adaptation*, in particular, are presented in Chapter 2. Here, we continue by summarising the contributions of this thesis, followed by a summary of the reminder of this document.

1.3.1 Contributions of the dissertation

A unified mediation framework. The main contribution of this thesis is the proposition of *a unified mediation framework* to achieve interoperability from application down to middleware layers. We rely on existing work in the domain of protocol mediation and complement existing approaches with a novel solution for the cross-cutting issue of *data adaptation*. More specifically, we devise a mechanism to *reuse existing message translators*, by composing them using a declarative solution, taking into account the data dependencies between the application and middleware layers. Our approach is more efficient than state of the art approaches in terms of development effort because we do not require hand-coded specification of message formats.

However, composed translators cannot be automatically integrated as part of a mediation framework (or any other system) without knowing the precise data models they use. By precise data models, we refer to the intermediary data types in which translators output parsed message data, and in which they expect data (that need to be composed into network messages). As far as we know, state of the art approaches in the domain of type inference [7–10] are unable to solve this problem.

To this end, we provide a formal mechanism, using tree automata, that generates an associated data-schema for an arbitrary translator composition. This contribution enables the inference of correct data-schemas, relieving developers from the time-consuming task of defining them. The inference type problem we solved addresses a specific class of data transformations which we call the *substitution class*. The provided inference algorithm is generic and can be directly applied in many other applications that require message translators and the ability to quickly adapt to new ones. Such applications include: Packet Analysers, Firewalls, Enterprise Service Buses, etc.,

AmbiStream middleware. A secondary contribution of this thesis is represented by AmbiStream, a lightweight middleware layer that complements the existing software stack for multimedia streaming on smartphones with components that enable interoperability. AmbiStream is a component of the CONNECT [11–15] protocol mediation approach extending it with the capability to handle "real-time" and "on-demand" multimedia streaming protocols.

Experimenting interoperability on real-world systems. As part of our research, we concerned ourselves with the applicability or our results. In this work we have included the results on two of our experiments.

- In Chapter 5, we present our proposition of a unified mediation framework using a real

world interoperability use case in the domain of conference management applications. We study the requirements and analyse the performance of our implementation using the web-services of Amiando (`http://developers.amiando.com/`) and Regonline (`http://developer.regonline.com/`) [1].

- In Chapter 3, we evaluate performance of on-the-fly data-adaptation for multimedia streaming protocols on current generation smartphones. Specifically, we evaluate the translation of the Real Time Streaming Protocol (RTSP) to HTTP Live Streaming (HLS) on Android and Apple devices.

Much of this work has already been published[2] in peer reviewed conferences, journals or as research reports. Here, we enumerate the most important:

- In [16] "Composing Message Translators and Inferring their Data Types using Tree Automata", we present the mechanism that enables reuse legacy message translators along with the algorithm for inferring data-schemas for composite translators.
- In [17] "A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet", we provide a unifying framework for protocol mediation that is agnostic to the interface-mapping applied, while integrating the up-noted contributions on message translator composition and data-schema inference.
- In [18] "AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices", we present the AmbiStream mobile middleware mentioned above.
- In [14,19] "Revised CONNECT Architecture" and "Final CONNECT Architecture", we extend AmbiStream to fully integrate with the CONNECT solution on protocol mediation. We also provide improvements towards dealing with mismatching buffering requirements when mediating between "real-time" and "on-demand" streaming protocols.

1.3.2 Structure of the document

The reminder of this dissertation is structured as follows.

- Chapter 2 serves as an overview and positions our contribution with respect to state of the art approaches in protocol interoperability. We describe with more detail the

[1] We have no affiliation or any other kind of agreement with either Amiando or Regonline. All trademarks and registered trademarks are the property of their respective owners.

[2] We acknowledge that some passages, definitions and figures have been quoted verbatim from articles that we published.

CONNECT approach towards making networked systems eternally connected which serves as both theoretical and technical basis for our contributions in the domain of protocol mediation.

- Chapter 3 takes into discussion a domain specific case of protocol interoperability, that is, the multimedia streaming class of protocols. Towards the objective of mobile deployed protocol mediators, we propose a design-time approach for enabling interoperability for the multimedia streaming protocols.

- Chapter 4 presents a method allowing the composition of legacy message translators followed by an approach towards the automated generation of abstract data-schemas for the process of protocol mediation. Specifically, this contribution enables the inference of correct data-schemas for composite message translators relieving interoperability engineers from the time-consuming task of defining them.

- Chapter 5 presents an unified mediation framework, complementary to the CONNECT enabler architecture, which supports mediator synthesis towards the goal of enabling *cross-layer protocol interoperability*.

- Chapter 6 concludes this work and discusses research questions that, in the authors' opinion, require further exploration.

Chapter 2

Background

Contents

2.1 Protocol interoperability

As we mentioned in the introduction, the difficulty of protocol interoperability 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., CORBA or SOAP) 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). For this reason, many approaches focus solely on achieving interoperability across the application and middleware layers assuming the use of IP at the network layer.

In what follows, we present a brief overview of the domain by underlying common state of the art solutions and highlight what they intend to solve.

2.1.1 Interoperable design by conformance

The most basic way to assure interoperability between components is to implement a compatible protocol on all sides. For more complex systems and systems-of-systems this task rapidly becomes difficult to manage. A common approach to overcome this challenge is conformance to protocol standards for implementing the means of interaction and information exchange between components. This compliance must be assured for entire systems from network to middleware and application layers. Standards Organizations like IETF, W3C, and ISO/IEC collaborate[1] to consistently define open standards for the Internet.

The existence of open standards led to the wide acceptance of certain protocols, and even allowed the emergence of convergence protocols. This is the case of the IETF Internet protocol suite (TCP/IP) which has become ubiquitous, in favor of other stacks/protocols like Novell IPX/SPX, ANSI-ITU Asynchronous Transfer Mode (ATM), AppleTalk, IBM SNA Data Link Control (DLC), etc. This tendency can be observed, for instance, from the continuously decreasing number of network protocol stacks supported by recent versions of Microsoft Windows OS[2].

On the one hand, protocol convergence confirmed that open standards are a well adapted solution for network protocols, like the ones part of the TCP/IP stack, and for special classes of applications like, for instance, *file transfer* (e.g., FTP, HTTP), *electronic communication* (e.g., SMTP, POP, IMAP, XMPP) and *infrastructure support* (e.g., DNS, BOOTP, DHCP).

On the other hand, when applications implement a more elaborate or business-specific application logic, which is likely to change frequently, defining standards is not helpful. Yet, applications can be built by reusing common components and their respective proto-

[1] `http://www.w3.org/2010/11/TPAC/W3C-IETF-Collaboration.pdf`

[2] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms739935.aspx`

col layers. For instance, message-oriented middleware (e.g., AMQP [20], JMS [21]) allow applications to be implemented on top of an abstraction that facilitates the communication and coordination of distributed components despite the heterogeneity of the underlying platforms, operating systems, and programming languages, thus facilitating, to some extent, system interoperability. However, middleware protocols also define specific message formats and coordination models, which makes it difficult for applications using different middleware solutions to interoperate.

2.1.2 Protocol conversion

On many occasions, legacy systems and applications have to be made interoperable after they were designed, implemented and deployed. The creation of protocol adapters is difficult as it often requires reverse-engineering and analyzing the systems in question. Protocol converters, as formalized in [22], are used to translate a standard or proprietary protocol of one component to the protocol suitable for the other component to achieve interoperability. This conversion includes conversion of data messages by rearranging data from an input message to an output message, and also adapting differences in the state-machines of the two protocols. Because implementing protocol converters is difficult and error-prone, tools like z2z [23] and others [24, 25] offer more suitable Domain Specific Languages (DSLs) for: (i) specifying protocol behavior (network interactions, synchronous or asynchronous messages, etc), (ii) describing the structure of the source and destination protocol messages (i.e., parser specification) and (iii) how the messages are translated between the source and the destination protocol. Z2z [23], in particular, combines a language for specification of protocols and messages, a compiler that automatically generates protocol gateways using C code, and a runtime that executes and manages protocol gateways. Z2z evolved into Starlink [26] which enables protocol translation dynamically at runtime, a particularly important feature in systems where existing protocols are unknown at compile time.

Protocol analysis methods, commonly used in the domain of Network Security, which require the specification of protocols, like [27], remarkably resemble protocol conversion techniques, with the mention that they focus only on silently processing the exchange rather than generating messages and triggering interaction with the systems under analysis.

2.1.3 Interoperability platforms

A recurring problem of protocol conversion methods is that a special convertor has to be implemented for each incompatible protocol pair. This is usually a concern when complex systems have to assemble an important number of legacy components and to access external services, each implementing *functionally-compatible* protocols. To address this constraint,

Nakazawa et. al. [28] propose a taxonomy of bridging solutions among communications middleware platforms and present uMiddle, a system for universal interoperability, which supports mediation (entities and protocols are translated to an intermediary common representation) and is deployed as an infrastructure-provided service. This design choice is appropriate for bridging communications middleware, since it requires communication through different transport technologies that may not be available on all nodes.

Common middleware approach 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 deployed on Mac, Windows, and Linux machines can seamlessly access a SOAP-based Web Service deployed on a Windows server. However, a CORBA client cannot access a SOAP-based 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 [29] 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 [6] facilitate this task, either by providing an infrastructure to translate messages into a common intermediary protocol or by proposing a Domain Specific Language (DSL) to describe the translation logic and to generate corresponding bridges [30].

Enterprise service bus (ESB). An evolution of the concept of protocol bridging is applied by Enterprise Service Buses [31]. An ESB represents an integration broker for heterogeneous systems such as web-services, applications, data-stores and devices. Interoperability is achieved through the means of an intermediary protocol (e.g., SOAP) and a common data representation format (e.g., XML). The message broker allows taking incoming messages from applications and dynamically applying transformations like routing, aggregation, decomposition and re-composition (e.g., IBM WebSphere Message Broker [3]). Depending on the ESB product, some integration tasks can be done on-the-fly (e.g., composition of service endpoints and management of service interactions), while others are done off-line since they require external implementation and deployment of new integration components (e.g., creation of a hardcoded adapter). All ESB-related solutions make the

[3] `http://www.ibm.com/websphere/wbimessagebroker`

assumption, however, that any service can be adapted to an intermediary communication abstraction represented by the message bus, which is not always possible or sufficiently efficient. We stress that the solutions above provide only an execution framework and still require developers to fully implement or specify the translations needed to enable the applications to interoperate.

Model driven architecture (MDA). MDA[4] proposes to specify applications using an abstract model, i.e., the Process Independent Model (PIM). The PIM is deployed atop middleware platforms described by the Platform Specific Model (PSM). This decoupling enables the modeling of application-middleware data dependencies, which may facilitate interoperability when used in relation with an interoperability architecture. However, MDA does not specify how to deal with heterogeneous PSM or PIM models, thus not solving interoperability. Yet, the existence of abstract system models (PIM) in relation to more concrete models (PSM), sets this solution at the boundary between *interoperability platforms* and *protocol mediation* techniques.

2.1.4 Dynamic synthesis of protocol mediators

All methods previously mentioned present a consistent flaw, that is, they require extensive human intervention, which is either for analysing the systems, designing the interoperability solution or for implementing or specifying the component which might take the form of an ad hoc wrapper, protocol converter, ESB adapter or protocol bridge. As described in [5], this is a problem of integration. Systems, application, and more generally, components should make explicit any assumption about interaction in order to make integration feasible and, possibly, automate it.

Solutions oriented toward dynamic synthesis of protocol mediators rely on intermediary entities, *mediators* [32], to enforce interoperability by mapping the interfaces of functionally-compatible components and coordinating their behaviours. Solutions for the synthesis of mediators [24, 33–38] 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 [39] 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. Furthermore, as we will later discuss in Chapter 5, when systems are multilayered it is also

[4] `http://www.omg.org/mda/`

Discovery	Discovering mediation candidate systems.
Compatibility checking	Assessing if the discovered systems are functionally compatible.
Building abstract system models	Identifying, learning and reasoning about system requirements including their interface and behaviour.
Mediator synthesis	Finding a valid coordination to mediate network interactions based on the systems' behavioural semantics. Autonomously validating the correctness of generated mediators.
Data adaptation	Finding the mechanism upon which messages are to be translated between the source and the destination protocol.
Deployment	Deploying a mediator while relying only on the local/accessible infrastructure.
Monitoring	Monitoring the mediator to assure it is functioning as expected.

Table 2.1 – Challenges of synthesising protocol mediators.

important to make explicit any cross-layer requirements, like for example, the way an application needs to configure a communication middleware layer to validate functional or non-functional (e.g., quality of service) requirements.

2.2 Protocol mediation: the CONNECT approach

Dynamic protocol mediators, as they were characterised by the CONNECT [11–15,19,40,41] vision for making systems eternally connected are oriented towards universal interoperability. This means mediators have to cope with multiple levels of system heterogeneity from middleware to application layers. In order to **synthesise concrete protocol mediators**, one must **discover**, **analyse**, **learn** and **reason** about heterogeneous networked systems. A synthesised protocol mediator must be **dependable**, **unobtrusive**, and **evolvable** while not compromising the quality of software applications.

2.2.1 Phases of protocol mediation

Indeed, the biggest challenge is synthesising such mediators. This vision implies the existence of the following components: (i) a software component capable of analysing individual systems and create protocol mediators, (ii) a supporting infrastructure for deploying and running mediators. Mediators accompanied by a supporting infrastructure should be able to overcome the challenges presented in Table 2.1.

Discovery. The mismatch of discovery protocols and mechanisms is a separate interoperability problem on its own. In the CONNECT project [14], the discovery of heterogeneous

networked systems is assured by an independent component (called Discovery Enabler) which implements a bridge for commonly used discovery protocols.

Compatibility checking. When they are discovered, one must assess if any of the encountered networked systems are functionally compatible. Of course, to automate this task, there is the need to define generic models for networked systems. Such models should be sufficiently detailed and include their role, interface syntax, behavior and non-functional properties. However, there is little consensus between state of the art solutions to this problem.

Building abstract system models. System models can be obtained during the discovery stage or, to some extent, inferred automatically. For instance, behavioral semantics may be extracted using automata learning techniques described in [42], provided that the interface syntax is known. However, most solutions, including CONNECT, require important specification effort for the creation of certain fragments (i.e., message parser specification, protocol state-machine, behavioral mapping and data mapping) of system models using proprietary DSLs. From this perspective, we may argue that there is little difference between **protocol mediation** solutions requiring significant low-level specification (i.e., complete specification of message parsers for all protocol layers) like Starlink [26, 30] and **protocol conversion** solutions discussed above. Authors usually claim a good balance between expressivity and ease-of-use when proposing new specification languages, but such properties are subjective and hard to measure in a consistent way. Relying on detailed models of functionally-compatible networked systems, mediators must reconcile the communication between networked systems at runtime, that is, to translate messages and data between the source and the destination protocol, as well as to implement a valid coordination to mediate network interactions based on the systems' behavioral semantics.

System behaviour. When dealing with the behavioural semantics of systems, some solutions [22, 27] require modelling system behaviour at each protocol layer (i.e., of each process/protocol composing an application's protocol stack). Other system models [15, 26, 30] maintain a global *Black-Box* state (also the case of CONNECT) of the system as it is observed from exchanged messages. But messages are not the only events which may trigger the system to change state. We have to acknowledge the existence of cases where user or system generated events change the state of a networked system. For example, mobile platforms support centralised out-of-band notification mechanisms (e.g., Apple Push Notification Service) using the server push communication paradigm. We can expect that a push notification may "wake-up" a mobile application to perform some processing. But

push communication is opaquely managed by the operating system together with a remote service operated by the platform provider. This kind of communication is, in general, inaccessible to a protocol mediator by common means, thus, requiring specialised integration with the mobile platform.

Mediator synthesis. The synthesis of protocol mediators has been the subject of a lot of work, as surveyed in [43]. Mediator synthesis is also known as interface mapping [15,24,44], and its result is referred to as a mediator or an adaptation contract [45,46]. Mediator synthesis establishes the semantic correspondence between the messages sent by one component and those expected by the other component. The work in [47] proposes a framework to formalise the process of synthesising CONNECTors that mediate two incompatible protocols, and suggests that data adaptation can be solved through ontology integration. To provide full automation, several approaches extract the interface mapping either by measuring the syntactic similarity of messages [44] or by verifying the semantic compatibility between their operations and data using ontologies [48]. In their seminal paper, Yellin and Strom [24] propose an algorithm for the automated synthesis of mediators based on predefined correspondences between messages. By considering the semantics of actions, Vaculín *et al.* [49] are able to infer the correspondences between messages automatically. To generate the application-layer mediator, they generate all requested paths and find the appropriate mapping for each path by simulating the provider process. Cavallaro *et al.* [50] 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 [36] 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.

Mediator synthesis at runtime 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. In [38], Bennaceur *et al.* have specifically developed a solution combining ontology reasoning and constraint programming to synthesise application-layer mediators at runtime.

Discovery	Singleton component. Discovers networked systems in face of discovery protocol heterogeneity. In collaboration with the learning enabler builds Networked System Models for each system, learns affordances, and matches networked systems based upon goals and intent.
Learning	Singleton component. The Learning Enabler uses active learning algorithms [42,51] to dynamically determine the interaction behaviour of a networked system and produces a model of this behaviour in the form of a labeled transition system (LTS).
Interaction	Singleton component. Uses the Starlink [26] tool to dynamically invoke networked system actions irrespective of the middleware protocols employed.
Synthesis	Singleton component. Takes the enhanced LTS models of a pair of networked systems and constructs [15,38] a CONNECTor model in the form of a k-Coloured automaton.
Deployment	Singleton component. Receives k-coloured automata from the synthesis enabler, and deploys this on a running instance of the Starlink tool; this can be on the same host as the Deployment enabler, or on a separate identified host within the network.
Monitoring	Singleton component. Receives monitoring data from probes instrumented in the CONNECTor. Forwards notifications to channels that other enabler's subscribe to.

Table 2.2 – Definition of the main CONNECT Enablers

Data adaptation. Data adaptation refers to the array of mechanisms and tools that enable mediation frameworks to translate messages between a source and a destination protocol. While some approaches have studied ways of adapting data directly between two protocols, it has become clear that a better approach consists of solving this problem in two phases as follows. First, messages are transformed into a common intermediary message format (or abstract format, as referred to by some authors). Secondly, a mapping between these data has to be found either manually, or automatically.

Data adaptation is a central topic of this thesis. However, to better underline the context and requirements of data adaptation, we further present the general process of generating mediators (aka. CONNECTors) in the CONNECT project.

2.2.2 The process of generating a CONNECTor

The CONNECT project is a concretisation of the concepts presented in the previous section, aiming at achieving protocol interoperability by relying on CONNECTors. CONNECTors are concrete emergent system entities, a specific kind of dynamically generated protocol mediators.

Further in this section, we provide a brief introduction, describing the process of synthesising CONNECTors. This process is based on a set of formal foundations which allow learning, reasoning about and adapting interaction between networked systems at runtime. As we show in Figure 2.1 and detail in Table 2.2, CONNECT is constructed in the form of multiple *Enablers* collaborating to produce on the fly *Mediators* between heterogeneous

networked systems. The architecture does not specify the place where the *Enablers* (such as *Discovery* and *Synthesis*) are deployed, but requires that communication between enablers is possible by the means of *Message Channels* and *Queues*.

CONNECTors are generated by the CONNECT process and then deployed between two networked systems to enable interoperation between them. Therefore, CONNECTors are a fundamental part of the CONNECT architecture. They are both produced and managed by the Enabler architecture (see Figure 2.1) and therefore their properties inform the key architectural principles of the Enabler architecture. They are also directly instrumented in order to validate QoS and security properties (i.e. communicating with the enablers that perform monitoring). Deployed CONNECTors are executed via interpretation by the Deployment Enabler which is based on the Starlink tool.

In Figure 2.1 we provide the configuration of the CONNECT Enablers at the connection phase. While the entire process is meaningful for achieving interoperability, given the context of this thesis we focus specifically on the Synthesis and Deployment phases which are the most relevant to our contributions. For this reason, we also include references for various models that are used during the Synthesis and Deployment phases, grouped as **Networked System Models** that are used during Synthesis and, respectively **Mediator Models** that are interpreted by the Deployment Enabler.

The dynamic synthesis and deployment of a CONNECTor is outlined by the following phases:

1. Discovery phase: Initially, the Discovery Enabler discovers networked systems (NS) available on the network. Systems must announce their presence using one of the several legacy discovery protocols supported by the discovery protocol bridge including DPWS and UPnP. For each NS, it stores the **Interface description** and performs an initial phase of matchmaking to determine which pairs of systems are likely to be able to interoperate. The matchmaking is based on **System Capabilities** which are manually specified and its associated **Domain Ontology**. The Interface Descriptions may need to be defined manually, since not all discovery protocols include them. Furthermore, these interfaces (independently on how they were obtained) need to be manually annotated with ontology concepts.

2. Learning phase: Whenever the system behaviour is unknown (usually because it is not advertised by the discovery protocol used by the system), such pairs of NS interface descriptions are passed to the Learning Enabler that uses active learning algorithms to dynamically determine the interaction behaviour and produce a **System Behaviour** model. The Learning Enabler will rely on the Interaction Enabler to communicate with the NS.

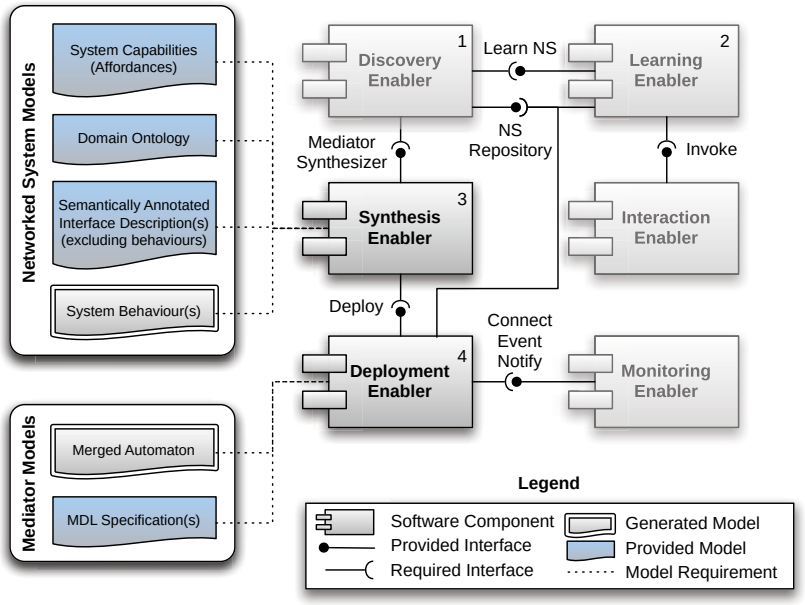


Figure 2.1 – The configuration of the CONNECT enabler architecture for the connection phase [14] including system models that are used either during the synthesis or the deployment of mediators.

3. Synthesis phase: When the Learning phase is completed, the pair of NS descriptions is then passed to the Synthesis Enabler that has the role of creating a mediator for solving i) application-level interaction model interoperability and ii) middleware-level interaction model interoperability. The LTS received from the previous phase is middleware specific, meaning that transitions are strongly correlated to the behaviour of the middleware protocol used by each system. The first step done by the Synthesis process is to extract a middleware-agnostic model of application behaviour (the detailed process of achieving this is presented in [52]). Then, a pair or application-level behaviour models are used to synthesise a correct-by-construction mediator using interface-mapping [15]. The LTSA (Labeled Transition System Analyser) model checker is used to generate the parallel composition of the mapping processes and to verify that the overall system successfully terminates.

4. Deployment phase: If the Synthesis Enabler is capable of producing a mediator. The mediator, which takes the form of a **Merged Automaton** is transformed into a k-coloured automaton, i.e., the format supported by the Starlink [26] tool. Along with the **Merged Automaton**, in order to deploy a CONNECTor, Starlink also requires a pair of **Message Description Language (MDL) Specifications** which specify

the network level message format used by each system. The MDL descriptions are provided externally, and are currently defined manually.

2.2.3 The Starlink framework

To deploy and execute a CONNECTor, the previously described models are interpreted and executed using the Starlink tool. That is, Starlink executes on a networked host in the environment; the model is loaded into Starlink, and when the networked systems begin communicating with another Starlink interprets the appropriate transitions in the concrete k-coloured automaton. A high level vision of a CONNECTor implementation is first presented in Figure 2.2; this illustrates the principle software elements that compose each CONNECTor and their overall behaviour.

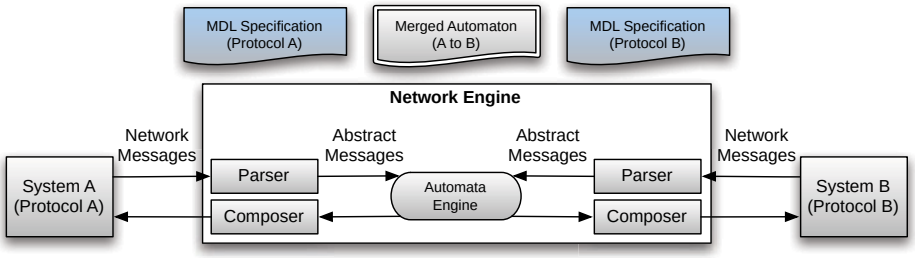


Figure 2.2 – Architecture of the Deployment Enabler [19] (based on Starlink framework [26]).

The **Network Engine** provides a library of transport protocols with a common uniform interface to send and receive messages. Hence, it is possible for a CONNECTor to receive messages and send messages using multicast (e.g. IP multicast), broadcast and unicast transport protocols (e.g. UDP and TCP) in order to directly communicate at the network level with the networked systems.

A **Parser** interprets the content of a distinct protocol message (or frame in a streaming protocol). That is, based upon the protocol's message format specification it reads the network data and produces a single Abstract Message instance; this is a uniform representation of network messages that is used by the CONNECTor to understand and manipulate the data. Parsers are generated from the description of a single protocol provided as a **Message Description Language (MDL)** document. For example, the MDL of SOAP messages is used to construct a parser that will parse SOAP messages. A full description of the MDLs developed in the CONNECT project, how listeners are generated, and how parsers execute is provided in [53] ("Realising Listeners and Actuators") and is not detailed further here.

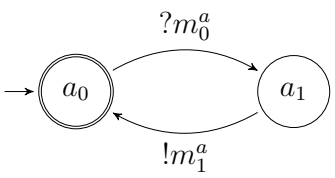
An **Composer** performs the reverse role of a parser, i.e., it composes network messages according to a given middleware protocol, e.g., the SOAP Composer creates SOAP messages. Composers receive the Abstract Message as input and translate this into the data packet to be sent on the network via the Network Engines. Like parsers, each composer is generated from a protocol's MDL specification.

The **Automata Engine** forms the central co-ordination element of a generated ConnecTor. Its role is to execute a mediator described as a merged k-coloured automaton, which documents how content received from one networked system (in the form of an Abstract Messages) is translated into the content and middleware messages required by the other networked system. Hence, the k-coloured automata mediator handles both application and middleware heterogeneity; it is able to address the challenges of: different message content and formats; different middleware protocol behaviour, e.g., sequence of messages; different application data formats; and different application operation behaviour. Each Mediator is specified in terms of merged k-coloured automata. The automata engine interprets and executes these automata directly.

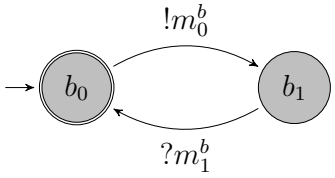
Definition 4 (k-coloured Automaton) *A k-coloured automaton, as defined in [26], is a deterministic finite automaton represented formally by* $\mathcal{A}_k = (Q, M, q_0, F, Act, \rightarrow, \Rightarrow)$*, where:*

- Q *is a finite set of states of a protocol.*
- M *is a finite set of either incoming or outgoing message types of a protocol.*
- q_0 *is the start state, that is, the state of the protocol before any message has been sent or received, where* $q_0 \in Q$*.*
- $Act = \{?, !\}$ *where ? is the receive action and ! is the send action.*
- $\rightarrow \subseteq Q \times Act \times M \times Q$ *is the transition relation that can be either a receive-transition or a send-transition. A transition relation has the form* $s_1 \xrightarrow{?m} s_2$ *for* $(s_1, ?, m, s_2) \in \rightarrow$ *and changes the state of the automaton from* s_1 *to* s_2 *once the message* m *is received. The latter is noted* $s_1 \xrightarrow{!m} s_2$ *for* $(s_1, !, m, s_2) \in \rightarrow$ *and changes state from* s_1 *to* s_2 *once the message* m *is sent.*
- $\Rightarrow$ *is a message history operator defined as* $\Rightarrow \subseteq Q \times Act \times \vec{m} \times Q$*. Where* $\vec{m}$ *denotes a sequence of stored messages.*

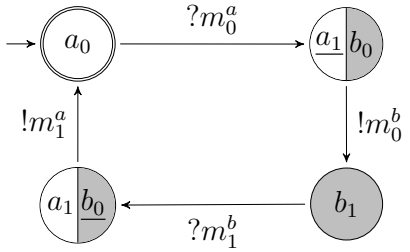
As we illustrate in Figure 2.3, two protocols A and B can be made interoperable if and only if their k-coloured automata $\mathcal{A}_a$ and $\mathcal{A}_b$ can be merged in such a way that there exists a state in $\mathcal{A}_a$ where the sequence of received messages is semantically equivalent to



(a) k-coloured automaton $\mathcal{A}_a$ of protocol A



(b) k-coloured automaton $\mathcal{A}_b$ of protocol B



(c) Merged k-coloured automaton of protocols A and B

Figure 2.3 – Example of k-coloured automata and their merged k-coloured automaton

the required output message in the initial state of $\mathcal{A}_b$. Furthermore, it is required that the sequence of received messages in the accepting state of the automaton $\mathcal{A}_b$ is *semantically equivalent* to the required output messages of a state of $\mathcal{A}_b$, that should lead to an accepting state of $\mathcal{A}_b$. In the simplest interoperability case, there will be only two bi-colour states, as it is the case in Figure 2.3c. However, for more complex protocols, it may be required to "switch colours" (i.e., to interact with both systems A and B in a sequence) multiple times before reaching a final state in A. Note that protocols A and B have compatible roles. For instance, in order to transition from state a_0 to the state a_1, the protocol A expects the reception of a message m_0^a, while the corresponding transition in protocol B entails the emission of a message m_0^b.

The precise mathematical relation of *semantic message sequence equivalence* denoted $\models$ is given in [26]. We only remind the basic definition. Informally, writing $n \models \vec{m}$ means message n can be constructed with information found in the sequence of messages $\vec{m}$. In other words, the semantic equivalence is true if and only if, for every field in the message n that is a mandatory field (as defined by the protocol), there exists at least one semantically equivalent field in the sequence $\vec{m}$. In the example above, the following relations must validate in order for the merged k-coloured automaton to exist:

1. $m_0^b \models (m_0^a)$ in state $\underline{a_1}|b_0$, meaning that m_0^b can be constructed using data from m_0^a.

2. $m_1^a \models (m_0^a, m_1^b)$ in state $\underline{b_0}|a_1$, meaning that m_1^a can be constructed using data from m_0^a and m_1^b.

Definition 5 (Semantic Equivalence) *A message n is semantically equivalent to the sequence of messages $\vec{m}$, denoted $n \models \vec{m}$, if and only if $\forall n \triangleright field \in \mathcal{M}_{fields}(n)$, $\exists m \in \vec{m} = \langle m_1 ... m_n \rangle \, | n \triangleright field \models m \triangleright field$, where:*

- *$\mathcal{M}_{fields}(n)$ is the set of mandatory fields of message n.*
- *$n \triangleright field$ is the operation to select a field from the message n. A field consists of a label and a type. The type can be either primitive (e.g., integer, string, char, etc.,) or a structured field when it is a composition of primitive fields.*

While the definition above sets the basic requirements for protocol data interoperability, it does not show how to solve it. Specifically, constructing messages using data contained in other messages is not trivial for the following reasons:

Message heterogeneity. Protocols use heterogeneous message formats.

Incompatible data models. Even when messages associated to a protocol use a homogeneous format (e.g., XML), they implement disparate data models and adhere to different data semantics.

Composite message formats. When applications use layered protocol stacks, message formats are also layered.

In order to address the up noted challenges, a number of questions must be answered. First, we must find the appropriate mechanisms through which messages are translated (i.e., parsed and composed). Second, we must give a more precise definition for the concept of *message* and how data/fields can be addressed inside a message (e.g., $n \triangleright field$). Third, we must find a solution to deal with composite message formats that contain data relative to multiple protocol layers. Last, we must investigate the means to specify or infer the mapping of message fields whenever semantic equivalence is validated. To clarify these questions with respect to state of the art approaches in protocol mediation, in the next section, we detail the process of *data adaptation*.

2.3 Data adaptation

We remind that data adaptation is a rather ambiguous term and can refer to the following problems: (i) mapping between concrete message formats, (ii) translation between concrete

message formats and common intermediary message formats, and (iii) mapping between common intermediary message formats. While the first is feasible in hand coded interoperability solutions, it is inefficient because a different implementation is required for each pair of message formats. For this reason, it is often desirable to abstract concrete message instances towards an intermediary message format for all systems. We refer to messages that were translated to an intermediary message format as *abstract messages*, or Abstract Syntax Trees (AST), knowing that structured data types are commonly represented as a hierarchy. Further in this section, we present the process of data-adaptation in a generic way rather than discussing aspects that are specific to the CONNECT project. However, for clarity, we make use of concepts such as *semantic equivalence* that were discussed above.

Network message. While it might seem trivial, it is important to clarify what we mean be a *network message* in the context of this work. A *network message* is a sequence of bits or characters which are sent over a network connection. We only consider messages starting from the Application Layer, meaning TCP/IP layer 5, or the OSI model layer 7. This is a weak assumption considering that most systems today use the TCP/IP stack as a convergence layer, or even higher middleware layers such as SOAP, HTTP, etc. The methods by which the streams of bits or characters are delimited into messages is a protocol specific or an application specific characteristic.

Network message layers. We mentioned in the introduction that whenever an application uses a multi layer protocol stack, the messages that are exchanged also consist of layers. While in the simplest cases these layers involve adding a header (supplemental data placed at the beginning of a block of data being transmitted) and possibly a trailer (supplemental data placed at the end of a block of data being transmitted), these layers are more often logical, meaning that at each layer data might be fragmented, multiplexed, transformed, etc. For this reason and in order to increase clarity, some works [54, 55] refer to a logical message layer as a Protocol Data Unit (PDU).

Message translation. Whenever a mediation approach intends to be technology-agnostic, messages must first be translated into an abstract representation. This is the task of *message translators*. They assure two functions: (message parsing) parsing a stream of bits or characters, representing a network message in order to produce a structured data representation which we refer to as an *abstract message*, and (message composition) processing an *abstract message* to produce a network message in the format expected by a given component. The two software components are often designed and implemented separately. However, we believe it is important to consider the two components in conjunction knowing

that they both have to agree on the use of the same *abstract message* format, as well as the same network message format. Message translators are either hand coded, hand coded using parser generators (e.g., Yacc, Bison, ANTLR), hand-coded using a Domain Specific Language [14,15,23,26,27,54–57], specified using an IDL (e.g., ASN.1 [58], Protocol Buffers, CORBA OMG IDL [59], etc.), or generated using a reverse engineering technique [60–62].

Abstract messages. There is little consensus about what an *abstract message format* should be. Nevertheless, significant differences can be found depending on the type of approach used to create message translators. Solutions oriented towards reverse engineering use the most rudimentary type of *abstract messages*. Specifically, solutions like [60–62] represent parsed messages as *Untyped Fields*, that is a sequence of untyped, and unlabelled blocks of data that can be either mandatory or optional. More advanced solutions in the field of network security (packet analysis and packet injection) [54, 55, 57] use *Composite Types* (such as the ones used in ASN.1 specifications). However, this expressive power comes at a cost. Such approaches require that *message translators* are hand-coded using a proprietary DSL, as opposed to being extracted automatically. Finally, solutions in the domain of protocol mediation [14, 15, 23, 26, 56] either support *Composite Types* using a proprietary encoding or use XML as the underlying language for *abstract messages* [14, 15, 26]. The use of the same format for all abstract messages is required in order to allow fields/data to be mapped from one message instance to another.

2.3.1 The process of data adaptation

To better underline the relation between the concepts introduced above, in Figure 2.4 we illustrate the process of data adaptation which involves a phase of abstraction and another phase of concretisation. Suppose that $m^b \models (m^a)$, meaning that the XML message m^b is semantically equivalent to the binary message m^a, and that m^b has to be constructed from the information contained in m^a. To support this, approaches like the ones presented in [14, 15, 23, 26, 56] commonly follow the following steps:

1. The message m^a is parsed and an *abstract message* a^a is created. While we show *abstract messages* formatted using XML, other formats can also be used. *Message translation* (steps 1 and 4) refers to the ability to parse messages from the network layer into an abstract message format that may be handled by synthesised mediators and then concretise back (a.k.a. compose / un-parse) the messages produced by the mediators into network messages.

2. A template of the destination *abstract message* a^b is generated using the *abstract message schema* s^b. An *abstract message schema* or *message grammar* is a document

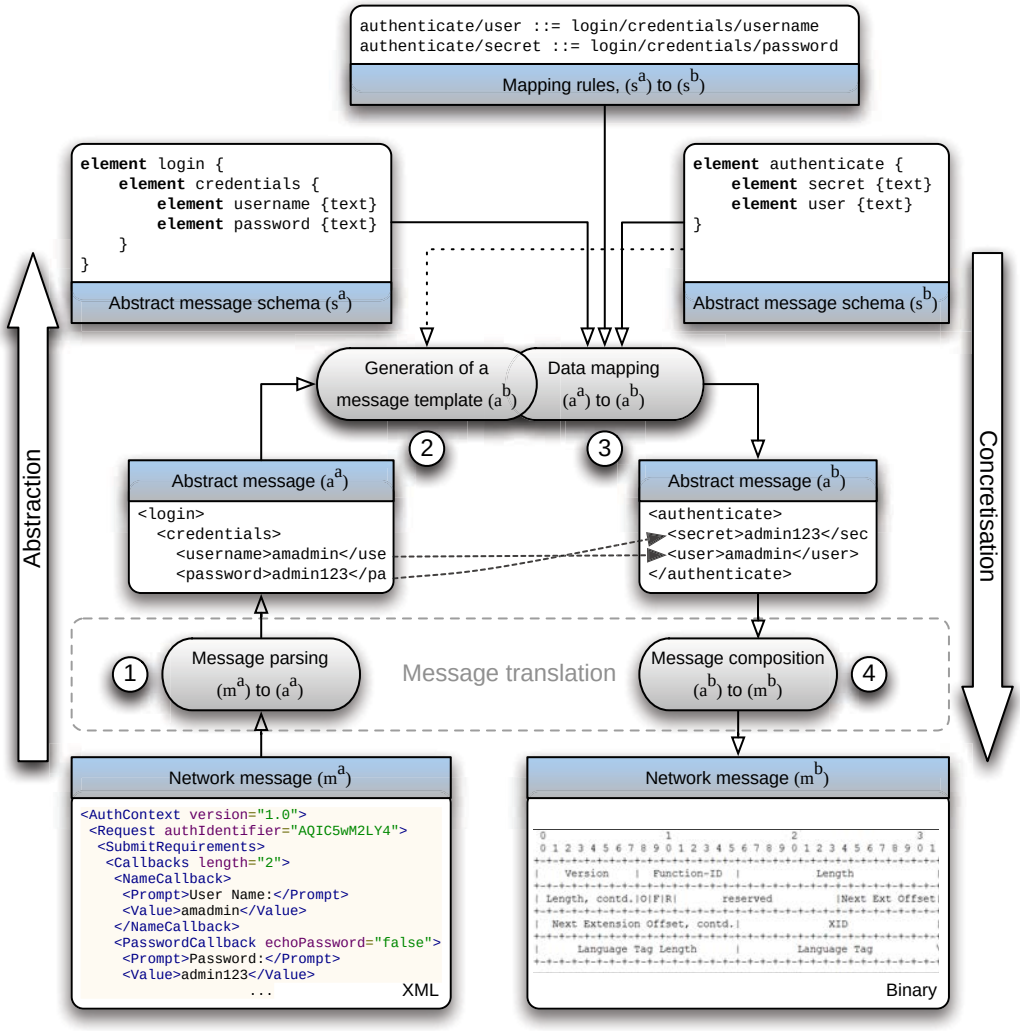


Figure 2.4 – Common phases of data adaptation in the field of protocol mediation. This figure intends to synthesise and generalise the data adaptation process present in the following publications [14, 15, 23, 26, 56].

specifying constraints on the structure and content of abstract messages, beyond the basic syntactical constraints imposed by the common intermediary format (i.e., XML in this case).

3. Using the *abstract message* a^a and the message template a^b, data can be mapped (i.e., assigned, merged, transformed, etc.,) from one message to the other. This is achieved by applying a set of *mapping rules*. Mapping rules define a transformation by which messages conforming to the *abstract message schema* s^a are to be transformed into messages conforming to the *abstract message schema* s^b. In order to specify or infer correct mapping rules between *abstract messages*, the *abstract message schemas* relative to both protocols part of the mediation must be available.

4. Finally, in order to produce a concrete network message, the last phase involves *composing* a^b into a message m^b whose format is specific to the destination protocol (in our example, a binary format).

2.3.2 Message translation

Message translation is one of the phases of data adaptation that cannot be easily automated. While automated reverse engineering techniques exist [60–62], the components they generate cannot be used along with automated protocol mediation solutions because they cannot fully characterise the abstract syntax of data contained within messages. In other words the *abstract message* formats they are able to extract are too basic to realise data mapping. Further in this section, we discuss common approaches to obtain message translators. Most existing approaches focus on the parsing problem, which is, in the general case, the hardest. In the following paragraphs we will use the word "translator" when we refer to approaches that solve both message parsing and un-parsing, and "parser" when we refer to approaches that address the parsing problem only.

There exist a plethora of approaches to build message translators: 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 translators are available also differ: translators can be precompiled components, or high-level descriptions using a domain-specific language. In Figure 2.5, we distinguish five classes of approaches to build message parsers.

Custom-made parsers and translators. These are components implemented in an *ad hoc* manner using a general purpose programming language.

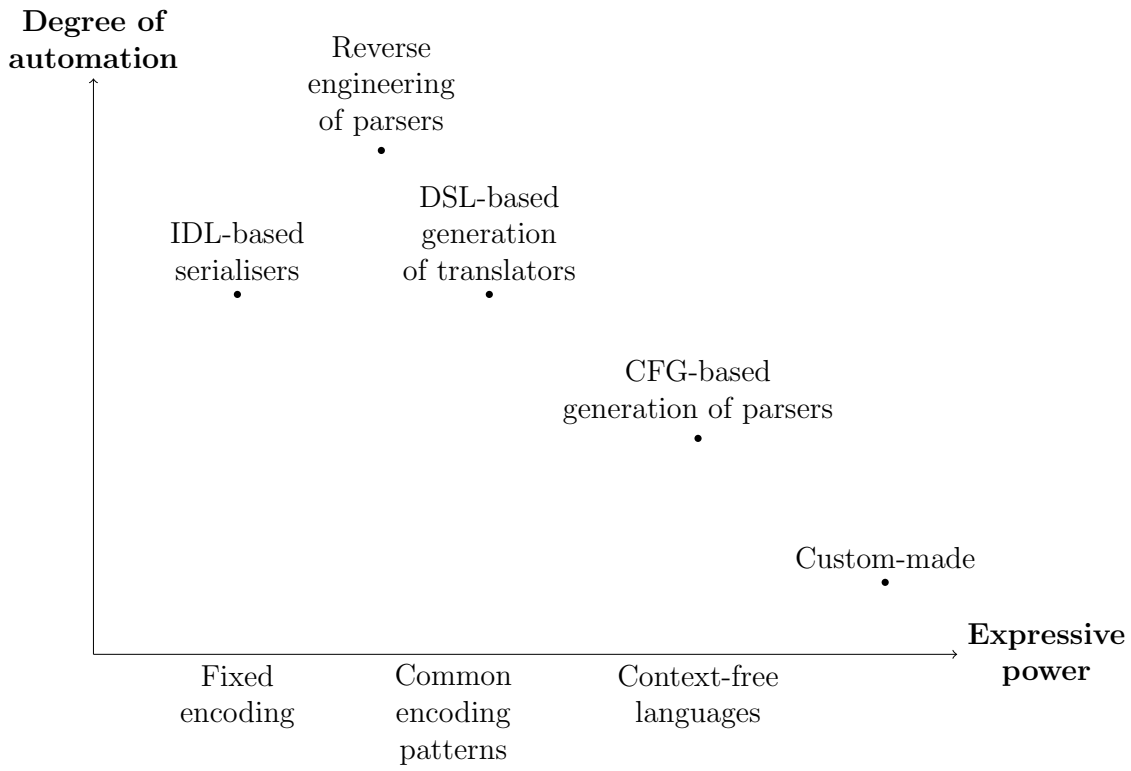


Figure 2.5 – Parser and translator design methods

CFG-based generation of parsers. An efficient alternative to the implementation of *custom-made* parsers is represented 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.

DSL-based generation of translators. DSLs can be used by experts to specify translators for complex message formats at a higher abstraction level, and in a more compact way, than CFGs (for parsers). Solutions for the generation of parsers and translators based on a DSL specification [23, 27, 39, 55, 56] 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 –TLV–, 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.

The Starlink [26] approach mentioned throughout this chapter proposes three indepen-

dent DSLs for text, TLV and XML message formats. Abstract message representations must be specified separately in the form of a sequence of fields of the form (name, data type, default value). The DSL for defining *Abstract Message Schemas* is rather redundant with XML Schema languages such as XSD and RelaxNG.

Z2Z [23] is a tool for generating protocol gateways based on a DSL for describing protocol behaviours, message structures, and the data mapping. Z2Z relies on the ZEBU [56] compiler for creating message parsers. ZEBU DSL syntax is very similar to that of ABNF. This approach also has the notion of *abstract messages* although they are called *message views*. Message views and are also defined manually using a secondary DSL. Abstract messages, as well as network messages are composed based on user-provided templates.

Message translation is also required in the domain of network security when implementing packet inspection software. GAPA [63], binpac [54], NetPDL [57] are tools that include a protocol specification language based on ABNF. The protocol specifications include message syntax as well as protocol states, transition logic, abstract message formats and encapsulation rules. While they are arguably more time efficient than using lower abstraction tools such as Flex (Fast Lexical Analyser) [64], Bison (GNU Parser Generator) [65], Yacc (parser generator) [66], etc., they only solve the parsing problem. All information including abstract message templates and message schemas must be manually specified by the user.

SCL [55] is a language based on ASN.1 for describing protocols and allows generating parsers and modifying input data at run time. Messages can be mutated and injected in a communication session in order to test protocol implementations. Abstract messages use a text based format called "Text Protocol Data Unit" (Text PDU). While SCL allows specifying abstract message formats using a syntax similar to ASN.1, it does not allow the specification of message parsers or message composers which have to be implemented separately.

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 abstract data format, they usually support a fixed (or, in the case of ASN.1, a small set of) message encoding mechanism. For this reason, 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 [59] 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.

Automated reverse engineering. Automated reverse engineering tools like Polyglot [60], AutoFormat [61] and Tupni [62] are capable of extracting message format specifications by analysing network traffic and by monitoring program execution. While they have a good record for identifying message fields for some message formats (exceeding 90%), they lack the ability to infer complex data types (abstract message schemas). It is worth mentioning that the tools mentioned above are only able to reverse engineer a small fraction of protocols existing today.

2.3.3 Data mapping

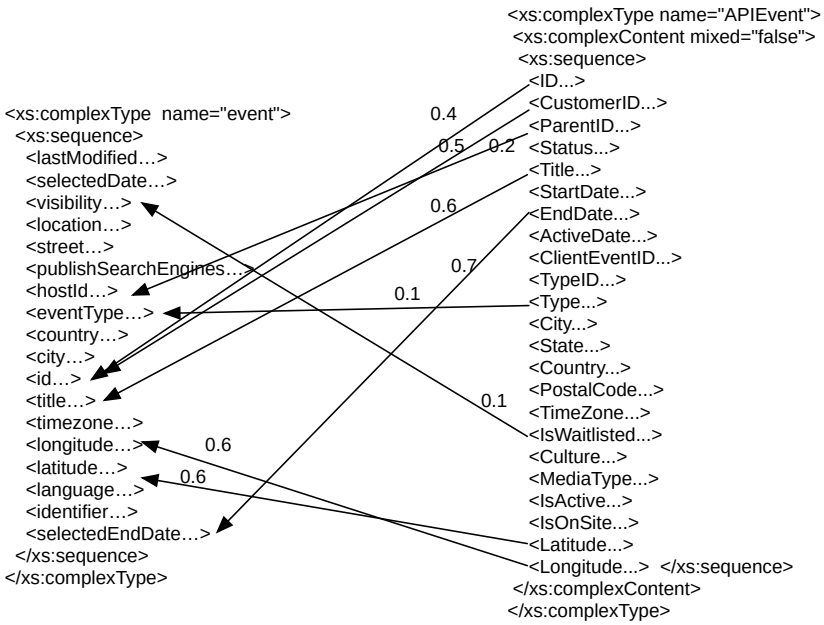


Figure 2.6 – Example of an XML schema matching result. The two abstract messages that are shown belong to applications in the domain of conference management. Each message encapsulated data relative to an event.

Message heterogeneity is not the only issue to be taken into account as part of data adaptation. Even when an abstract message format is agreed upon, data may not match

in terms of labels, scope, granularity of abstractions, temporal units, domain definitions and data-types [32]. To enable incompatible components to interoperate, the data need to be converted in order to meet the expectations of each component. Approaches in protocol mediation that focus on a specific technology [67] realise data mapping directly on the network messages whenever the formats in use allow this (e.g., the case of web services where data is encapsulated using XML). Whether if data *mapping rules* are specified manually like in [23, 26] or synthesised automatically like in [14, 15] they require that abstract messages are attached to an *abstract message schema* that defines constraints on the structure and content of *abstract messages*. Abstract message schemas also allow generating valid abstract message templates (or candidates).

Schema-Based Matching approaches [44, 68–71] allow finding *mapping rules* automatically when given a pair of *abstract message schemas*. The mapping is based only on syntactic similarity between formats and, in some cases, it is augmented by the semantic similarity of field labels. However, such solutions cannot be applied directly on systems because there is no guarantee that the data mapping rules with the highest similarity score is correct. We give an illustrative example of schema matching using the Harmony tool [5] in Figure 2.6. Notice that some fields with a high match score are indeed correct (e.g., 0.6 for `title` vs. `Title`), while others are obviously incorrect (e.g., `visibility` vs. `IsWaitlisted`; intuitively event visibility has no relation to the presence on a waiting list).

Besides estimating the similarity of attribute names or types, more knowledge of the domain is necessary to calculate complex correspondences. More advanced approaches [14,15] rely on ontologies [48] to formalise the knowledge about the domain. Ontologies build upon a sound logic theory to enable reasoning about the domain based on an explicit description of domain knowledge as a set of axioms [72]. Ontology reasoning allows generating a set of data mapping rules that are based both on the syntactic structure of messages as well as the semantics of data. However, such approaches require developers to annotate *abstract message schemas* with references to concepts in an ontology. In the CONNECT approach this is achieved by using Semantic Annotations for WSDL and XML Schema [73] to annotate abstract system interfaces with references to concepts in an OWL Ontology [74]. The provided ontology must be specific to the domain of the application (e.g., video streaming, conference management, medical, etc.,).

[5] Harmony is a open source schema matching tool part of the OpenII information integration tool suite `http://openii.sourceforge.net/`

	Approach Name	Message Translators	M.T. Expressive Power	Abstract Message Templates	Abstract Message Schemas	Schema Expressive Power	Data Mapping Rules
Interop.: Protocol Mediation	Zebu [56] with z2z [23]	H/C using DSL (Zebu)	Text, TLV+	H/C using DSL (z2z)	H/C using DSL (z2z)	Composite Types	H/C using DSL (z2z)
	Starlink [26]	H/C using DSL	Text, XML, TLV+	H/C using DSL	H/C using DSL	Composite Types	H/C using DSL
	Schema-Based Matching [44, 68–71]	N/A	N/A	Generated	H/C XML Schema	XML Schema	Generated (syntactic mapping)
	CONNECT [11–15] with Starlink [26]	H/C using DSL	Text, XML, TLV+	Generated	H/C SAWSDL [73] + H/C OWL Ontology [74]	XML Schema	Generated (syntactic and semantic mapping)
Security: Packet Inspection	GAPA [63]	H/C using DSL	Text, TLV+	N/A	H/C	Primitive Types	N/A
	binpac [54]*	H/C using DSL	Text, TLV+	N/A	H/C using ASN.1+	ASN.1 [58]	N/A
	NetPDL [57]*	H/C using DSL	Text, TLV+	N/A	H/C using DSL	Composite Types	N/A
	SCL [55]	H/C using DSL	TLV+	N/A	H/C using ASN.1+	ASN.1 [58]	N/A
	Polyglot [60]*	Generated	Text, TLV+	N/A	Generated	Untyped Fields	N/A
	AutoFormat [61]*	Generated	Text, TLV+	N/A	Generated	Untyped Fields	N/A
	tupni [62]*	Generated	Text, TLV+	N/A	Generated	Untyped Fields	N/A

Table 2.3 – Summary of approaches that solve various phases of data adaptation.

Legend:

N/A	not applicable
H/C	hand coded
TLV+	the class of type-length-value binary message encodings and extensions
Text	the class of "line based" text message encodings (e.g., HTTP, FTP etc.)
XML	the class of XML message encodings
ASN.1+	extension of the Abstract Syntax Notation One (ASN.1) language [58]
*	the approach only solves the parsing problem

2.4 Summary

In this chapter, we introduced the general context of Protocol Interoperability. Further, we presented the CONNECT approach that allows synthesising protocol mediators. Mediators are a means of dynamic protocol interoperability at run time. Last, we discussed in more detail the challenges and solutions for an important phase of *Protocol Mediation*, that is *Data Adaptation*.

Data formats, and in particular message formats, have long represented a barrier to interoperability [12]. This is because software parts often make different assumptions about how data is represented [75]. Table 2.3 summarises the solutions that solve various aspects of data adaptation. We notice that all of the proposed solutions require extensive amounts of hand coded (H/C) elements (models, schemas, specifications, etc.,), which in our opinion is contrary to the goal of automated protocol interoperability. We also notice that many of the solutions solve only parts of the data adaptation problem. For instance, all approaches in the domain of *Security:Packet Inspection* do not address the problem of *data mapping*. In this thesis we address these limitations as follows.

Streaming protocol interoperability. In Chapter 3 we present an extension to the CONNECT mediation approach that can solve interoperability for multimedia streaming protocols on current generation smartphones. Interoperability of streaming protocols is particularly challenging as opposed to other types of applications because of the following reasons. Real-time streaming protocols impose strict temporal constraints on the processing of packets, and, in order for multimedia content to pass over packet networks buffering mechanisms must be implemented. Furthermore, multimedia protocols have specific requirements for data adaptation which were not yet addressed by interoperability approaches. Specifically, each streaming protocol uses a multimedia container format along with message formats that are specific to the control part of the protocol. The *smallest common abstraction* for application data is a *primitive type field* (as we discussed earlier in this chapter). However, multimedia data must be transformed into a different common abstraction, referred to as *elementary stream samples*. To allow this, special message translators that are specific to multimedia container formats must be used.

Software reuse approach for translating composite message formats. A cross-cutting challenge that we address in this thesis is *protocol binding*, i.e., the way protocols are combined to form a protocol stack. Binding causes systems to exchange messages combining multiple syntaxes (i.e., composite message formats). The resulting message formats can include a mix of text encodings, binary encodings and data serialisation formats. Ex-

isting solutions to protocol interoperability have not fully succeeded in dealing with the increasing heterogeneity of components because of one 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 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 extensive and low-level hand-coded models and specifications.

To address the problem of translating composite message formats, in Chapter 4 we present a novel approach for composing pre-compiled message translators. This solution enabling software reuse, is complementary to approaches that allow generating message translators for single layer message formats. A major challenge in composing translators (and in turn, parsers) is represented by the way parsers are designed. Specifically, "parsers are so monolithic and tightly constructed that, in the general case, it is impossible to extend them without regenerating them from a source grammar" [76] Even if the source grammars are made available, composition is still an issue, taking into account that combining two unambiguous grammars may result in an ambiguous grammar, and that the ambiguity detection problem for context-free grammars is undecidable in the general case [77].

Mediation solutions usually rely on analysing abstract message syntax either to asses if systems are functionally-compatible or use abstract message syntax as constraints when building a mediator [24, 44, 48, 49]. Also, when systems rely on multi layer protocol stacks, application data may be scattered over different message encapsulation layers. To assure that the composite message translators we generate can be used in protocol mediation approaches, also in Chapter 4 we propose a mechanism capable of automatically generating data-schemas for composite abstract messages. On a more general note, the provided inference algorithm can be adapted to a number of applications beyond the scope of this work, such as XML Schema inference for a constrained set of XSLT transformations.

A unified mediation framework. In Chapter 5 we integrate our contributions as part of a unified mediation framework to achieve interoperability from application down to middleware layers.

Chapter 3

Interoperable multimedia streaming on mobile platforms

Contents

The following chapter presents work that has been done between 2011 and 2012. It proposes a mediation solution for multimedia streaming and it evaluates its effectiveness on mobile platforms (Android and iOS). Because the two mobile platforms are in fast and

continuous evolution, some technical aspects have changed since we initially published this work. While they are important, they do not eliminate the usefulness of our approach. There are two important technical advancements that have been made:

- API access to hardware and software video codecs. Access to system provided video encoders and decoders is of critical importance for multimedia streaming interoperability. This is because third party applications (including protocol mediators) can encode and decode video using highly optimised system components. On iOS, the API for the H264 video codec was added with the release of iOS 8 in September 2014. On Android, the API for accessing the VP8, MPEG4 and H264 system codecs was added with version 4.1 (Jelly Bean) in 2012. However, we found that the implementation of this API by device manufacturers only reached a satisfying level of robustness and compatibility starting with version 4.4 of Android (released in 2014).

- Widespread adoption of the WebRTC standard. Recently, WebRTC has became the de-facto standard for peer-to-peer multimedia streaming between web browsers. With the exception of Safari, WebRTC is now part of most notable desktop web browsers including: Firefox, Chrome and Opera. Concerning mobile systems, WebRTC can only be found on Android. However, many Android and iOS applications use an application-embedded version of WebRTC to handle peer-to-peer multimedia streaming. We further present WebRTC, as part of background, in Section 3.1.

The present generation of smartphones enables a number of applications that were not supported by previous generation cellular phones. Particularly, the greater processing power, better network connectivity and superior display quality of these devices allow users to consume rich content such as audio and video streams while moving. Not surprisingly, radios[1] and television channels[2] today provide mobile applications that allow access to their live media streams. Even video rental services[3] provide mobile applications that support movie streaming to smartphones.

All those applications, however, assume a centralised architecture where a powerful server (or a farm of servers) provide streams to lightweight mobile devices. Node heterogeneity also remains an issue: most of those applications are available for a single smartphone platform. Indeed, to support multiple phone platforms, developers must (i) modify the mobile application to support different sets of decoders, streaming protocols and data formats and (ii) generate multiple data streams on the server side to be consumed

[1] `www.npr.org/services/mobile`
[2] `www.nasa.gov/connect/apps.html`
[3] `itunes.apple.com/us/app/netflix/id363590051`

by each mobile platform. Hence, when a resourceful server is not available, as in the case of a peer-to-peer (P2P) streaming scenario, this approach is impractical.

Interoperability of multimedia streaming protocols among heterogeneous mobile platforms can benefit to a large number of applications, such as:

- (Broadcast) Streaming a live event directly to other devices reachable on the network;
- (Screencast) Sharing media on the fly between different devices (phone to tablet/TV);
- (VoIP) Voice call applications;
- (Mobile games) e.g., mixing augmented reality with live remote user interaction;
- Distributed processing of a video stream;
- Audio/video sharing in crisis situations when infrastructure is unavailable;

Today, to create such applications, developers must overcome a number of constraints. First, smartphones run different mobile operating systems, each supporting a different set of audio/video encoders, decoders and streaming protocols. Second, communication is performed over wireless networks that are unstable and that do not support resource reservation, and thus streaming quality is managed by the protocol without cooperation from the network layer. Finally, the multimedia streaming software stack of each platform is highly optimised to deliver high quality audio and video while reducing resource usage.

Existing system support for multimedia streaming is unsuitable to face the challenges described above. Indeed, architectures for multimedia streaming on the Internet such as [78, 79] suppose the existence of powerful servers that can adapt content on-the-fly on behalf of clients, which is infeasible when the streaming server is a resource-constrained smartphone. Solutions for multimedia streaming on ad hoc networks such as those surveyed in [80] require direct cooperation between application layers and networking layers, e.g., integration between the video codec and the routing protocol to optimise streaming quality. This approach is not appropriate on smartphone platforms since replacing the native software is sometimes impossible but often undesirable for performance reasons.

Live streaming protocols, either real-time or non real-time, commonly use two communication flows: one for assuring *Stream Control*, and another for *Media Transport*. To enable multimedia streaming among heterogeneous devices, the following challenges must be solved:

Interoperability stream control protocols. Multiple incompatible protocols for multimedia streaming exist today, and each platform supports one or a small subset of

them. As a result, smartphones must overcome the streaming protocol heterogeneity problem to be able to exchange multimedia streams with heterogeneous devices.

Assuring the validation of timing constraints. Another requirement in order to enable interoperability between streaming protocols is to manage timing from two perspectives: first, real-time streaming protocols impose temporal constraints on the arrival and inter-arrival of packets, and secondly, streaming protocols manage flows of data, and in order for the content to pass over packet networks buffering is required. Buffering techniques differ from one protocol to another, and it is thus the role of the interoperability solution to solve this type of heterogeneity.

Adaptation media transport data formats. Each smartphone platform generates and stores multimedia data using some specific container format. These data cannot be directly transmitted through a different streaming protocol because the media container format is specific to the protocol. Smartphones, then, must also adapt the media container format to enable translation from the native streaming protocol to non-native protocols supported by other peers.

Implementing protocol translators for each existing streaming protocol implies a high development effort given the important number of protocols and mobile platforms. To address this limitation, we propose AmbiStream, a lightweight middleware layer, as an extension to CONNECT [14], that complements the existing software stack for multimedia streaming on smartphones with components that enable interoperability. CONNECT enables automated mediation between different protocols, but does not take into account the challenges introduced by mobility and particularly multimedia streaming on mobile platforms.

AmbiStream provides multimedia streaming interoperability amongst heterogeneous mobile devices with the following assumptions: (i) both the source and the destination support a common pair of audio/video codecs and (ii) the codec pair used is compatible with the destination's (client-side supported) streaming protocol. Multimedia transcoding is not necessary in most cases since a small set of encoders/decoders are available on most mobile platforms and are compatible with many of the existing streaming protocols. For example, the video codec H.264/MPEG-4 AVC (ISO/IEC 14496-10 / MPEG-4 Part 10, Baseline profile) is supported on Android (RTSP), iOS (HLS), Windows Phone 7 (IIS MSS) and Blackberry (RTSP).

The remainder of this chapter is organized as follows. In the next section we review existing work on multimedia streaming in mobile environments. In Section 3.2 we detail the challenges involved in creating a layer to adapt multimedia streams in mobile heterogeneous

environments. Section 3.3 presents the architecture of the AmbiStream layer and explains how the main components operate: the format adapter, the protocol mediator and the local media server. Section 3.4 discusses our experimental results on Android and iOS devices, which show that it is possible to adapt data and protocols at run time and also obtain streams with satisfactory quality.

3.1 Background on multimedia streaming

3.1.1 Interoperable streaming standards

Today's options for designing interoperable multimedia applications by conformance (to a well established standard) reduce to a single option, known as Web Real-Time Communication (WebRTC). WebRTC was released in 2011 by Google as an open source cross-platform library for browser-based real-time communication. The project was later picked up by the World Wide Web Consortium (W3C), who released an API draft [81]. The idea behind the project is to evolve into a universal media streaming standard that would eventually be included as a base component of mobile as well as desktop Web browsers. Based on WebRTC API, application developers would be able to create browser-to-browser applications for voice calling, video chat, and P2P file sharing without the need of external plugins. This support can be naturally extended to native mobile applications, knowing that most, if not all, mobile platforms allow developers the integration of Web view components inside applications[4]. However, this standardisation effort is still ongoing (as of November 2015), and legacy applications not conforming to this universal standard would still be unable to interoperate. Along with the API for managing multimedia sessions, WebRTC is also integrated with a set of network management tools, known as the Interactive Connectivity Establishment (ICE) framework, that allow applications to open and maintain data streams in a peer-to-peer fashion while circumventing network restrictions such as firewalls and NATs. Indeed, with today's mobile devices that constantly switch networks to allow peer mobility and also support different types of underlying communication infrastructures (e.g., Cellular, Wi-Fi, Bluetooth), it is rather problematic to open and maintain uninterrupted streams of data. However, as standardisation efforts cannot keep pace with the rapid development of new technologies, better interoperability approaches have to be found.

[4]A Web view is a user interface (UI) application component that displays web pages. On most platforms Web views include methods to navigate through a history, zoom in and out, perform text searches, etc.

3.1.2 Streaming middleware and protocol mediators

Many multimedia-oriented middleware have been proposed in the literature. One of the earliest efforts in this direction was proposed in [82], which provided applications with mechanisms for late binding based on QoS constraints. The proposed platform was later extended in [83] to leverage CORBA's mechanisms for inspection and adaptation and enable applications to adapt the stream quality based on information obtained by inspecting middleware components. However, as predicted in [84], the lack of mature multimedia support at the middleware level led the industry to develop platform-specific solutions to handle multimedia streaming quality. As a result, today, most existing streaming protocols integrate mechanisms to adapt video quality to network conditions. Other middleware solutions have been proposed to provide multimedia streaming services. Chameleon [85] is a middleware for multimedia streaming in mobile heterogeneous environments. It is implemented using pure Java Core APIs in order to be portable to all Java and JavaME handsets. In Chameleon, servers send streams with different levels of quality to different multicast groups, so that clients can select the best quality according to their available resources and also adapt to changes on resource availability by selecting a multicast group providing a stream with lower quality. This approach imposes a heavy burden on the server side, which has to keep multiple streams in parallel regardless of the number of clients. Furthermore, Chameleon implements the whole software stack required for streaming, which has a negative impact on performance.

Fewer works take into account the capabilities of current smartphones and their impact on mobile multimedia streaming. The evaluation of streaming mechanisms in [86] for Android 1.6 and iOS 3.0 tries to identify which design is better suited for mobile devices. Traditional metrics such as bandwidth overhead, start-up delay and packet-loss are used to evaluate the quality of multimedia streaming in various test situations. They observe that high network delays can result in non-continuous playback when using the HTTP Live protocol from iOS, while RTP streaming remains unaffected on Android. Even though this work does not provide a solution for solving heterogeneity issues between these two platforms, provided results must be considered when designing a mobile streaming framework.

3.2 Challenges of mobile interoperable streaming

Towards the goal of enabling peer-to-peer streaming of multimedia data between heterogeneous smartphones, we further discuss the following challenges: (i) how to enable interoperability among incompatible streaming protocols, and (ii) how to adapt media containers to consume multimedia data transmitted through an incompatible streaming protocol.

Here, we detail the challenges introduced above. Specifically, Section 3.2.1 reviews the process of streaming multimedia data from a server to heterogeneous clients. Then, based on this general schema, Section 3.2.2 details the challenges involved when translating multimedia streaming protocols, while Section 3.2.3 explains the issues caused by the different media container formats available on current smartphones.

3.2.1 The streaming process

Streaming to heterogeneous devices is classically done by servers supporting a set of audio/video *codecs*, *media container formats* and *streaming protocols*, and comprises three phases: *media capture*, *media transmission* and *media presentation*.

Media capture. Media content can originate either from a camera, stored data or from a remote source via a streaming protocol. Possibly the most important characteristic of a multimedia content is its audio/video encoding. Indeed, being a highly resource demanding operation, multimedia encoding is subject to software and hardware optimisations on both personal computers and embedded devices. The availability of encoders and decoders therefore varies depending on the mobile operating system, platform and device. If a client does not support a decoder compatible with the server's encoder, the client cannot consume the media. When a server supports multiple encoders, multimedia data can be transcoded into a format compatible with the client supported decoders, but this process is resource consuming and can affect performance, especially when streaming live content. Transcoding also impacts image and sound quality and introduces additional latency for live streams.

Media transmission. Since video and audio frames cannot be directly transferred over an IP network, they are encapsulated within media containers that provide the necessary meta-information to facilitate the decoding and correct presentation on the receiver (i.e., client) side. The process of wrapping and unwrapping audio/video frames from a media container is also referred to as multiplexing and demultiplexing, respectively. This is related to the fact that in some container formats, frames (or frame fragments) from multiple audio and/or video tracks are interleaved. The media transmission also requires control and signalling. This task is assured by means of a communication protocol specifically designed to transport multimedia content.

Streaming protocols can be divided into two subgroups:

Real-time protocols are best suited for conversational content such as video conferences where user interaction with the streamed content is important.

On-demand protocols are designed to offer better scalability and connectivity; it is usually based on the higher level Hypertext Transport Protocol (HTTP) and introduces acceptable delays.

Media presentation. In order to correctly reproduce an audio/video stream on a mobile phone, it is required that the platform supports the given streaming protocol, media container format, the audio/video codecs (and, for some codecs, the codec profile used by the encoder). Being a resource consuming activity, multimedia decoding is usually managed by the mobile platform through hardware decoders or by efficient native code implementations. To offer a satisfactory multimedia user experience on resource-constrained devices, mobile platforms provide a shared application component for multimedia playback (i.e., a *media player* component) that applications can access through a standard API. This approach has the advantage of providing a uniform multimedia experience regardless of applications. However, it limits the possibilities to improve audio/video handling in mobile devices since the exposed API is generally limited. For instance, existing decoders used by the player to display multimedia content might be inaccessible for use or extension by applications.

3.2.2 Heterogeneity of protocols and signalling mechanisms

Most smartphone platforms support at least one streaming protocol client. The most well known protocols used in mobile phones today are: Real Time Streaming Protocol (RTSP) [87], Apple HTTP Live Streaming (HLS) [88], Microsoft Smooth Streaming[5] and Adobe HTTP Dynamic Streaming (HDS)[6] (provided that the mobile platform supports Adobe Flash). The most commonly found on mobile platforms is RTSP, but because it uses UDP as transport protocol on unprivileged ports it is inappropriate for use in restricted networks such as 3G and public WiFi hotspots, or connected to a Gateway performing Network Address Translation (a case where incoming connections are not possible due to lack of IP reachability). A standard extension defined in [89] enables interleaving messages over the TCP control connection, but is not supported by many implementations. Protocols designed for video-on-demand scenarios, such as HLS and HDS, are almost equivalent in terms of functionality and concept, but differ in message formats and media containers.

Real-time streaming protocols are generally designed over the UDP transport protocol because timeliness is much more important than the reliability offered by TCP. Consequently, simple reliability features, such as sequence numbers, sequence identification, syn-

[5]http://www.microsoft.com/silverlight/smoothstreaming/
[6]http://www.macromediastudio.biz/products/httpdynamicstreaming/

Protocol / Codec	H.263	H.264	MPEG-4	AAC-LC/AAC+	AMR-NB	MP3
RTSP	$\mathcal{A},\mathcal{B}$	$\mathcal{A},\mathcal{B}$	$\mathcal{A},\mathcal{B}$	$\mathcal{A},\mathcal{B}$	$\mathcal{A},\mathcal{B}$	–
RTSP interleaved	–	–	–	–	–	–
RTSP - SRTP (secured)	–	–	–	–	–	–
HTTP Live Streaming	$\mathcal{A}$	$\mathcal{A},\mathcal{I}$	$\mathcal{A},\mathcal{I}$	$\mathcal{A},\mathcal{I}$	$\mathcal{A}$	$\mathcal{I}$
HLS over SSL (secured)	–	$\mathcal{I}$	$\mathcal{I}$	$\mathcal{I}$	–	$\mathcal{I}$
MS Smooth Streaming	–	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$
MSS over SSL (secured)	–	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$

Table 3.1 – Streaming protocols versus audio/video decoders supported on mobile platforms (as of December 2011), where $\mathcal{A},\mathcal{B},\mathcal{I},\mathcal{W}$ symbolise Android, Blackberry, Apple iOS and Windows Phone 7 mobile operating systems.

chronisation codes, continuity counters, flags and timestamps are integrated in the media container layer to cope with the unreliable nature of the transport. Such features are not necessarily found in the same configuration in all formats. As a result, transforming a real-time stream to a video-on-demand fragment requires complex buffering and efficient transformation of real-time data. Such requirements impose strict temporal constraints for the transformation.

Still, even if the streaming protocols are incompatible by default, the encoded video and audio elementary streams may be compatible with multiple devices. For example, HLS on Apple's iOS uses H.264 codec for video, but the same codec is also largely used to stream video over RTSP to Android devices. As it can be seen in Table 3.1, there exists a common set of video and audio decoders available on multiple mobile phone platforms. In contrast, streaming protocol support is increasingly heterogeneous, with the arrival of new proprietary protocols such as HTTP Live Streaming and Microsoft Smooth Streaming. Thus, we conclude that multimedia data can be exchanged between heterogeneous smartphones without the need to perform costly transcoding operations. However, it is still necessary to adapt streaming protocols to enable streaming between heterogeneous devices.

3.2.3 Multimedia container adaptation

The conversion between different media container formats is a critical requirement for assuring interoperability between heterogeneous streaming protocols. Supporting both real-time and video-on-demand protocols makes this task more complex due to the mismatching of properties of the protocol groups.

Encoded elementary multimedia data is stored on disk using a media container format (e.g., 3GPP, MP4, AVI). Such containers are designed to be used only in random access scenarios. Therefore they are not suitable for streaming over a network connection. Another type of containers are streamable media containers (e.g., MPEG-TS, ASF, PIFF). They are

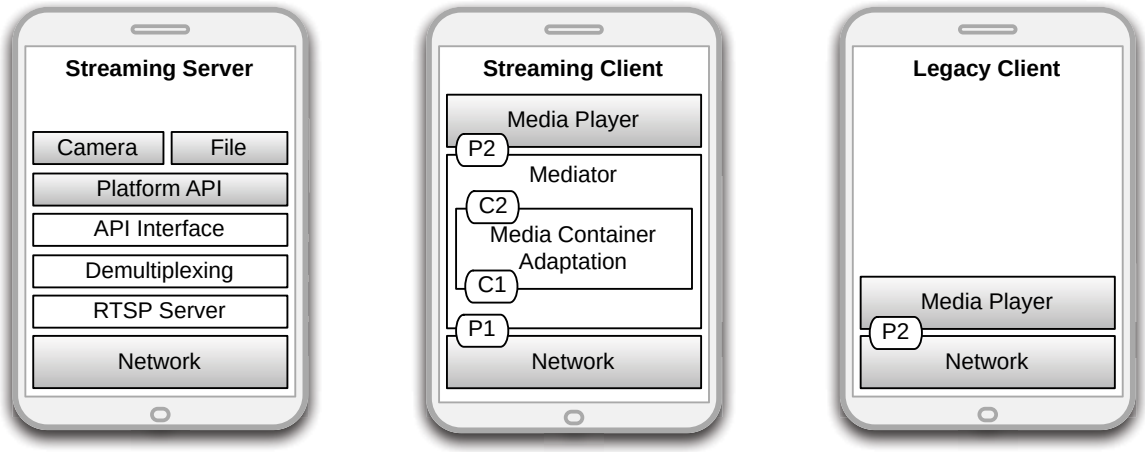


Figure 3.1 – The AmbiStream middleware architecture

designed to be transported over IP packet networks, provide methods for fragmenting audio and video streams and may also offer synchronisation and recovery mechanisms to cope with network delays or packet losses. The encapsulated media packets can contain multiplexed audio/video tracks (e.g., MPEG-TS, PIFF) or single tracks (e.g., RTP). Depending on the streaming protocol type (real-time/on-demand), multimedia fragments differ in size and structure. In general, real-time protocols use lightweight headers and small packet sizes, usually less than the MTU[7] in order to reduce the transfer delay by avoiding packet fragmentation. Video-on-demand protocols regularly use large video fragments composing 10-30 seconds of audio/video each. Such formats commonly rely on the ISO base media file format[8] structure which supports storing of multiple interleaved frames inside a single fragment, [90, 91]. Larger fragments reduce the need of receiver buffers but also introduce a start-up delay which is at least equal to the duration of the first fragment.

3.3 AmbiStream architecture

The overall architecture of AmbiStream is presented in Figure 3.1. It includes a compile-time *Mediator* which can be deployed on the streaming client device. AmbiStream enables multimedia streaming protocol interoperability (extending the approach proposed by Starlink [26]) in two directions: *first*, it enables the translation between real-time and on-demand streaming protocols, which requires buffering, dropping and combination of

[7]Maximum transmission unit (less than 1500 bytes for Ethernet)
[8]ISO/IEC 14496-12:2008

messages to deliver time-sensitive data at the right intervals, *second*, we support adaptation of container formats, which in the case of multimedia are dependent on the streaming protocol. Both operations are done by the *Mediator*, which is specified by the means of high level models using a set of Domain Specific Languages.

Streaming server. On the server side, a platform specific *API interface* is used to access the *Camera* data stream, or a *File* from memory. The stream data is then demultiplexed into elementary stream tracks (e.g., the audio track). Then, the data-stream is passed to a mobile RTSP server that we implemented.

Streaming client. The client receives the *Media description*, and instantiates the appropriate protocol mediator (e.g., *P1* to *P2*) and media container adapter. Streaming protocol mediators and media container adapters (e.g., *C1* to *C2*) are used as pluggable components created at compile time. To simplify support for a large array of protocols, these components are generated from descriptions of messages and protocol behaviour given in the form of DSL (domain specific language), as detailed in Section 3.3.1. The *Media container adapter* is further detailed in Section 3.3.2. Depending on the adapted protocol, the samples might be buffered at this point.

Legacy client. The *Mediator* can also solve interoperability for legacy streaming-enabled devices, which do not allow software (or firmware) extensions (e.g., televisions). In this case, the Mediator is not deployed on the server or on the client, but on another smartphone which we call an "AmbiStream mediator support node". We have successfully tested this technique using Android smartphones as *server* and *mediator support node*, and an array of phones, running on different platforms, as legacy devices (including iPhone 3G, Nokia N8, Sony-Ericsson W715, etc).

The AmbiStream architecture enables smartphones to stream multimedia between each other without involving a third party server, since all the adaptation is performed on the client side. In terms of privacy, this solution is superior to other architectures that require the stream to pass through an untrusted server for adaptation and/or distribution. Even legacy clients, that receive the streaming from an intermediary node instead of directly from the server can select a trusted peer based on any trust establishment protocol.

3.3.1 Streaming protocol mediation

Our approach is inspired by Starlink [26], a run-time solution for protocol interoperability. Although on the fly mediation, as supported by CONNECT, is more flexible and en-

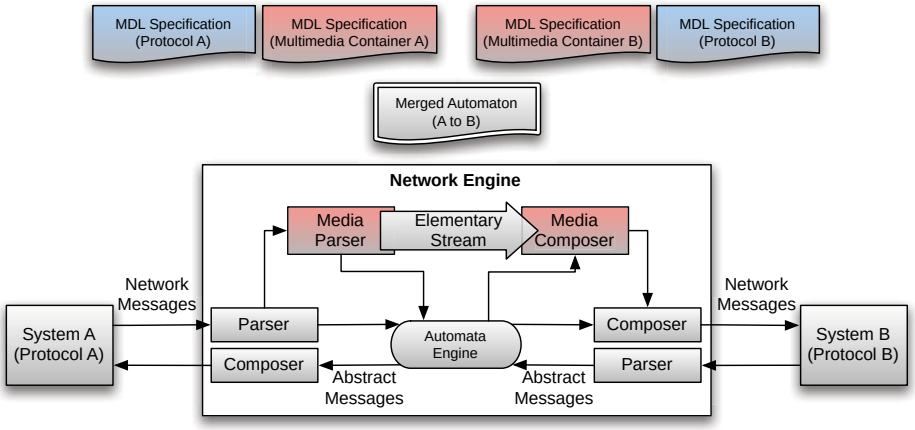


Figure 3.2 – Starlink architecture extension for live streaming

ables adapting protocols that are unknown at compile-time, in our case we decided to use compile-time mediators, given the resource-restrictions of mobile devices. In addition, the availability of a specific protocol is only subject to the support of mobile platforms, thus making it possible to know in advance the adaptation requirements of each mobile device. We thus propose a compile-time interoperability solution based on Starlink.

The mediator is specified by a developer in the form of three DSL-based models, as illustrated in Figure 3.2:

- **Protocol message format DSL** (MDL Protocol A/B) which is based on the Starlink MDL, describes the format and structure of message fields of the streaming client protocol. This model is used to synthesize message parsers for incoming messages and composers for outgoing messages.

- **Multimedia container format DSL** (MDL Container A/B) is used to specify the multimedia container format used by "Protocol B". It is different to the first language, as it is specifically designed to capture aspects of live multimedia streaming, by adding support for common operations such as message timing, fragmenting and multiplexing.

- **Merged automaton DSL** (Merged Automaton A-to-B), is used to specify the behaviour of the Mediator in the form of a k-coloured automaton. Although currently used in relation to an intermediary protocol (RTSP), the DSL is not restricted to this usage.

The models are passed on to a compiler that produces multi-language (Java, C and C#) Mediators.

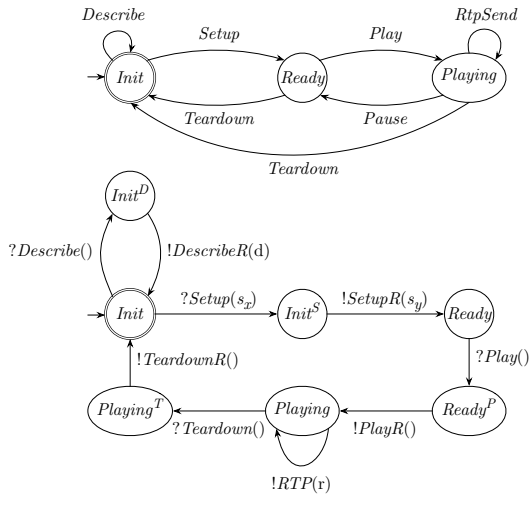


Figure 3.3 – Simplified LTS describing the behaviour of an RTSP Server

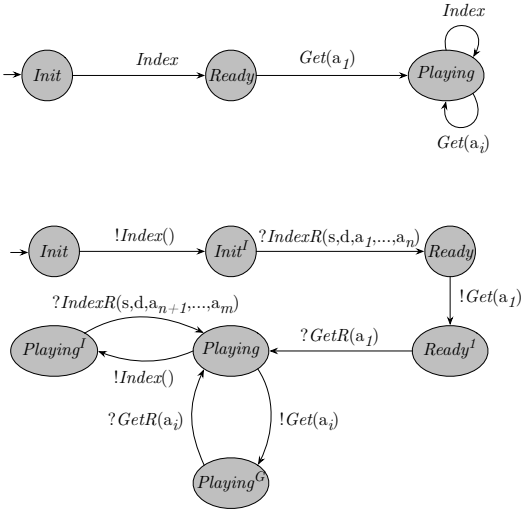


Figure 3.4 – Simplified LTS describing the behaviour of an HLS Client

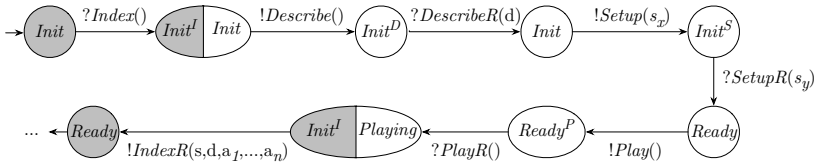


Figure 3.5 – Fragment of the k-Coloured Automaton of the RTSP-HLS mediator

Further in this section we motivate the rationale behind our design solution based on a concrete example of live streaming protocols: RTSP and HLS. Then, we approach the challenges introduced by the chosen example. We choose Real Time Streaming Protocol [87] as the source (or server) protocol and HTTP Live Streaming [88] as the client. The simplified behaviour of the RTSP server is presented in the top part of Figure 3.3 and the behaviour of the HLS client in the top part of Figure 3.4 in the form of labelled transition systems. As it can be easily observed, although the two protocols are quite different in design, the first being real-time and the second high-latency, the application states are quite similar. Both protocols mainly follow a request-response messaging pattern, thus each transition must be duplicated in request and response actions. This is shown in the bottom part of the two figures, and further detailed below. The two automata share the same definition of k-Coloured Automata we presented in Section 2.2.3.

RTSP Server: The RTSP server receives a *setup* message from the client meant to describe the client's connection request. The server responds with a *setup* response containing information such as a session ID and transport ports. While in *Ready* state, the server can receive a *play* request. For simplicity we chose the TCP interleaved transport mode, where multimedia packets are sent in-band over the same TCP connection as the control messages. In *Playing* state, the server will constantly send one-way RTP messages. The stream is interrupted when the server receives a *teardown* message. RTP is a lightweight wrapper for audio or video samples. RTP messages are sent at a frequency relative to the content sampling rate. The RTP packet frequency is not equal to the sampling frequency because large samples are fragmented in multiple RTP messages sharing the same *Timestamp*.

HLS Client: The HTTP Live Streaming Protocol uses HTTP as a transport protocol for both session control and stream data transport. The basic message flow of this protocol is as follows: the client application sends an HTTP request to download an extended M3U playlist (*! Index()*). The playlist received contains (i) a sequence number, (ii) a list of stream *chunks* (i.e., $a_1, a_2, .., a_n$) and (ii) a chunk duration, representing the playtime in seconds of one chunk. The client starts downloading stream chunks in order (*? Get(a_i)*). Intuitively, HLS is actually breaking the overall stream into a sequence of small HTTP-based file downloads. When all downloads are complete, the client requests a new, updated *Index*, from the server. This process can be repeated, supporting, in this manner, unbounded streams.

Analysing protocol heterogeneity. A number of important differences with respect to multimedia streaming can be observed between the two protocols:

1. HLS file chunks are much larger than RTP messages, and thus contain a greater stream duration. Common values are 5 to 30 seconds for a HLS audio/video chunk, and around 30 milliseconds for an RTP video packet[9].

2. HLS is not a real-time streaming protocol. Live streams are delayed by at least three times the duration of a chunk (e.g., 3 x 5 seconds) and in case of network congestion, this delay can further increase (this delay can be bounded by server configuration by reducing the number of buffered chunks).

3. The description of the transported stream in terms of: audio/video codec, decoder profile, sampling frequency and channel identification are provided as a text descrip-

[9]Configurations commonly observed in existing applications

tion for RTP (using Session Description Protocol (SDP), IETF Proposed Standard as RFC 4566/2006). For HLS such information is not available at protocol level, but can be extracted from the MPEG-TS multimedia container used to wrap the stream chunks.

4. The HLS client uses a request-response pattern to obtain stream data while RTSP uses one-way RTP messages.

5. An HLS client requires a large media buffer (which should be available for download from the server in the form of chunks, e.g., 15 seconds) before starting presentation, while RTSP only requires a small buffer (usually <1s) on the client side in order to eliminate packet inter-arrival jitter.

MDL Specification. The message description language we propose is very similar to the one proposed by Starlink to generate parsers and composers of protocol messages. Data values contained within message fields are transformed into primitive types or sequences of primitive types. On the one hand, such DSLs are useful when messages are of low syntactic complexity (e.g., text, XML). On the other hand, multimedia container formats are more complex, and such languages lack the expressive power to define them. Furthermore, as observed with concrete protocols there isn't a clear separation between application data formats and protocol message formats. In the worst scenario, the two layers are highly interleaved.

An example of a message description is shown in Listing 3.1. The description is divided in *Input* and *Output* to differentiate between incoming messages that should be parsed into structured data types and outgoing messages that are composed. This distinction is more important with text protocols, where messages have loose requirements in terms of line order, optional parameters, delimiters, spacing characters and so on. The DSL proposed here supports protocols that use either binary, text or XML message formats. To assure a sufficiently expressive message description, we extract the required fields using value capture patterns defined using Posix regular expressions for text protocols, XPath for XML and based on field size and location for binary protocols. The choice of Posix regular expression for text protocols was driven by its availability on most of the platforms, most notably that it is part of the GNU C library and is compatible with the regular expressions integrated in Java standard library (java.util.regex).

In the particular case above for RTSP to HLS (see Figure 3.6), the stream description is provided at protocol messaging level (using SDP) for RTSP and at the application data level (inside the MPEG-TS multimedia container) for HLS. To better illustrate this case, we consider the data field identifying the audio (or video) codec used in a streaming session

```
<Protocol type="text">
  <Input>
    <Header name="http_head">
        <Var name="Url" type="String"/>
        <Rule test="capture_order(Url)">1</Rule>
        <Capture var="Method"> [RegEx] </Capture>
        <Finish test="empty_line"/>
    </Header>

    <Message name="GET_IDX">
        <Insert>http_head</Insert>
        ...
    </Message>
    ...
  </Input>
  <Output>
    <Message name="IDX">
       <Var name="$TargetDuration" type="Integer"/>
       <Line>#EXTM3U</Line>
       ...
    </Message>
    ...
  </Output>
</Protocol>
```

Listing 3.1 – DSL describing message formats for the HLS protocol

for RTSP and HLS. For RTSP this information is in the form of a string-encoded field ("H264") of the *Describe* message. In the case of HLS, the same codec is identified by the one byte code "$0x1B$" of a Packetized Elementary Stream Message, encapsulated inside a sequence of MPEG-TS Packets and further contained in an HTTP response message.

To address this cross-layer dependency, in Figure 3.2 we suggest an improved framework structure for Starlink, by adding two more abstract models for multimedia container formats, and an extension to the *Automata Engine* to allow the transfer of messages between the application data and the protocol layer.

In order to support the definition of such models, further in this section, we present the AmbiStream *Multimedia Container Format* DSL. This language can be used to specify the application data formats of multimedia streaming applications. As seen in Figure 3.2, one multimedia container description will be used to generate a *Media Parser* for incoming stream data packets (in the case presented above, RTP), and another for the *Media Composer* of the second protocol's data format (MPEG-TS). While multimedia flows are only transmitted in one direction, applications such as Videoconferencing need to manage two flows in opposite directions. This requires to mirror the newly introduced components as a

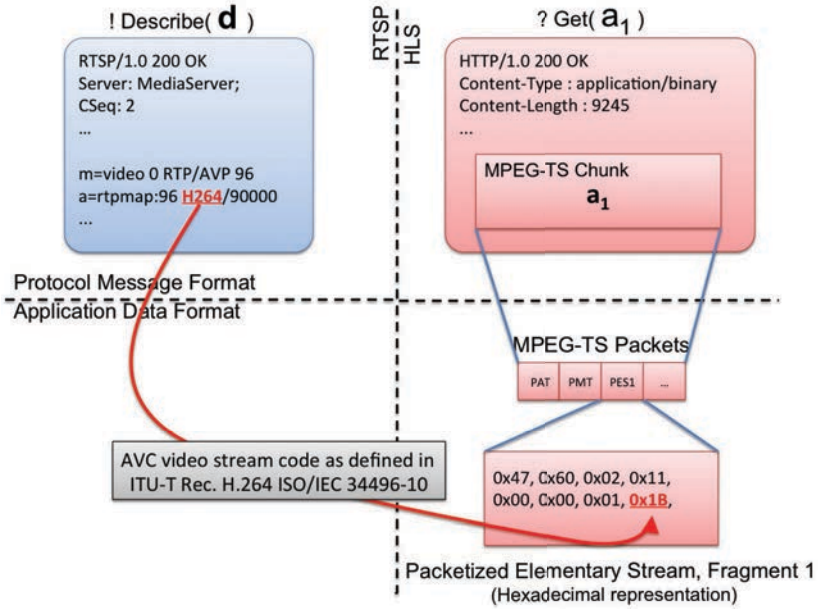


Figure 3.6 – Example of cross-layer message field mapping

means to add a second flow. In the case of multimedia container formats there is no need to synthesise a mapping between the input messages of the multimedia container format and the messages of the output multimedia container format. This is mainly because any multimedia stream can be transformed to a common elementary format called *Elementary Stream*. An *Elementary Stream (ES)* as defined by Moving Picture Experts Group[10] is usually the raw output of an audio or video encoder. In Figure 3.2 the multimedia flow is shown in the form of an arrow labelled *Elementary Stream*. Going back to the concrete example above, while both the server and client are in *Playing* state, the *Media Parser* should produce audio/video samples at the content sampling frequency (e.g., PCM audio encoding at 8,000 Hz). The *Media Parser* should also pass all stream description data to the *Automata Engine* to be used at protocol message level, and the *Automata Engine* should as well return multimedia specific information to the *Media Composer*.

3.3.2 Media container format adaptation

Translating the control part of streaming protocols is not sufficient to distribute multimedia between incompatible protocols. The format in which audio/video content is encapsulated also differs depending on the protocol. To achieve a complete solution, the translation

[10]http://mpeg.chiariglione.org/

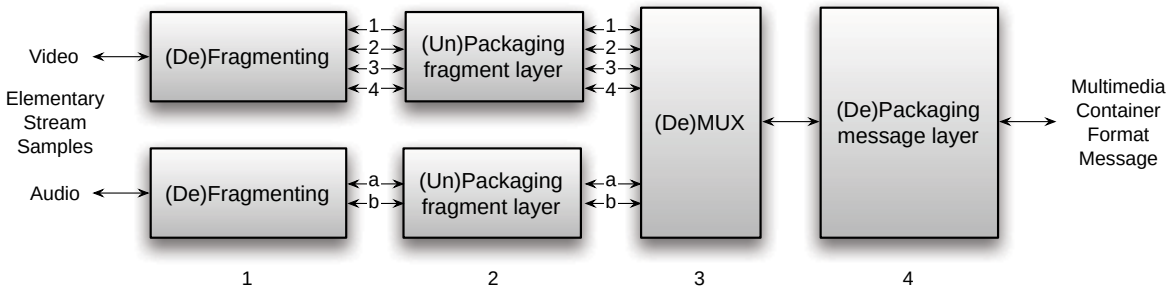


Figure 3.7 – Adapting the media container format

between media container formats must also be taken into account. The most important factors that led to the decision to separate this part from the protocol translation model are: the much higher complexity of multimedia packets, the dependence relation between messages (order, timing, fragmenting), the buffering requirements, and the multiplexer logic required to interleave multiple media tracks inside one packet/message.

We further divide the media container adaptation in four distinct steps: sample fragmenting, fragment packaging, multiplexing and message packaging. The process of adapting a stream composed of two tracks (one audio and one video) is sketched in Figure 3.7. Each of the four phases is defined by the developer using a DSL to describe multimedia containers, different than the ones used for protocol description. Similarly to the generation of protocol translation plug-ins, the description of the multimedia container adaptation is compiled to be deployed to designated platforms. To simplify the description, a number of media packet-related parameters are exposed through the DSL. Parameters include: the length of the media payload, media encoding, fragmenting flag, sampling frequency, sequence number, inner frame sequence number and first/last fragment flag. The components for protocol description and container adaptation are considered to be independent, thus allowing, for example, a protocol to choose between multiple supported data formats. The *Media Container Adaptation* middleware component (see Figure 3.1) parses real-time input and produces elementary stream samples for audio and frames for video. These samples are then composed into the destination multimedia container format.

Because we may use a real-time protocol for transporting multimedia data, the problem of timing should also be taken into account. To do so, we add a time-stamp reference to each packet resulting from any of the four phases of media format adaptation. Fragments of one frame share the same time-stamp information, while messages composing multiple frames contain the time-stamp of the first frame and their duration. The time required for a frame to pass through all of the phases required by the format, should not exceed the sampling interval of the content. Failing to assure this property can cause the client to

run out of buffered data, resulting in playback stalls. In order to prevent such behaviour, frames are intentionally dropped such that the output of the conversion is completed at the right time to assure a fluent playback.

The fragmenting step defines the way large audio or video samples are divided into smaller segments according to the limits imposed by the streaming protocol, by the media container or by the network configuration. For instance, in the case of MPEG-TS, the samples are split into fragments which are inferior in size to 184 bytes, such that they can be correctly contained inside the standard 188 byte packets. For RTP, fragmenting follows the standard RTP Payload Format depending on the codec used (for instance, the one described in [92] for the H264 video codec). The fact that protocols like RTP encapsulate elementary samples differently depending of the codec used, leads us to believe that using a modular Domain Specific Language, like the one we are proposing, simplifies the task of enabling interoperability between multimedia container formats.

In the case where media content is composed of multiple tracks (i.e., one video and one audio track), two separate fragmenting units are used. The number of fragments created from single frames is variable. Each fragment contains a reference to the time-stamp of its originating frame. The time required for fragmenting one frame should never exceed the sampling interval of the content. In case this requirement can not be respected, the quality of the stream is degraded by dropping frames.

In Listing 3.2 we give a fragmenting description for the RTP Container Format and in Listing 3.3 we give another fragmenting description for the MPEG-TS Container Format. In both cases, the description begins with the assignment of two attributes: (i) the `trackid` which is a unique identifier for a multimedia track, knowing that a stream can include multiple audio, video and data tracks, and (ii) the `type` of the track which is currently limited to the values `audio`, `video` and `data`. Further, the description includes a sequence of fragmenting methods. We support two methods: `data_format` and *data_length*. The first allows fragmenting one video frame or one audio sample depending on a data pattern. This is the case in Listing 3.2, where a H264 frame may contain multiple Network Access Layers (NAL) units. Each NAL unit is prefixed by the sequence `00 00 00 01` (given here in hexadecimal). A second method of fragmenting frames, which we call `data_length`, sets a maximum size for each fragment. This kind of fragmenting is useful when the multimedia container format uses a fixed package size, as it is the case for MPEG-TS or when limiting the packet size can increase throughput. The latter is commonly used for the RTP format and allows RTP messages which are sent over UDP to avoid datagram fragmentation (i.e., one fragment can be sent in one transmission unit of the underlying network protocol stack). The `value` attribute of a `data_length Method` element specifies the value to be used for determining the fragment size. The most frequent case in practice

is `fragment_packaged_length` which signifies the length of a fragment after the packet headers were added at the *fragment packaging* step. Because these headers can be variable in size, a developer is unable to specify this value statically. Notice that in Listing 3.2 we combine the two methods and apply them in sequence based on the `order` attribute.

```
<Fragmenting trackid="1" type="video">
 <Method type="data_format" order="1">
  <Block name="nal_unit">
   <!-- A H264 frame may contain multiple NAL units -->
   <Field name="nal_unit_start" value="0x00000001">
    <InputRange startbyte="0" bytelength="4"/>
   </Field>
   <Field name="nal_unit_type">
    <InputRange startbyte="4" bytelength="1"/>
   </Field>
   <Field name="nal_unit_data">
    <InputRange startbyte="5" bytelength="+"/>
   </Field>
  </Block>
  <ContentRule type="repeat($nal_unit)" value="+"/>
 </Method>
 <Method type="data_length" value="fragment_packaged_length" order="2">
  <!-- Conforming to a Maximum transmission unit (MTU) of
   1400 bytes to avoid datagram fragmentation. -->
  <ContentRule type="less_than()" value="1400B"/>
 </Method>
</Fragmenting>
```

Listing 3.2 – RTP Container Format. Fragmenting phase for the H264 codec.

```
<Fragmenting trackid="1" type="video">
 <Method type="data_length" value="fragment_packaged_length">
  <ContentRule type="less_than()" value="188B"/>
 </Method>
</Fragmenting>
```

Listing 3.3 – MPEG-TS Container Format. Fragmenting phase.

The frame layer packaging stage adds/parses individual packet headers. This transformation conforms to [87] for RTP packets and [93] for MPEG-TS. Depending on the protocol, the resulting packets are passed to the multiplexer or sent directly to the protocol translator. In Listing 3.4 we give a fragment of the DSL description for the packaging phase of the RTP container format and a H264 video track. Each field of a *fragment layer* packet is described using the `Field` element. A field includes a name, a template value,

an `InputRange` and optionally a `Rule`. The template `value` allows setting the data of a field when messages need to be composed, but it is also used when parsing messages in combination with a selection `Rule`. The `Rule` element is a rather generic component of the DSL. At the packaging phase we are particularly interested in `selection` rules, which allow defining different parsing/composition strategies depending on various parameters. For instance, in Listing 3.4 line 24 the template value for the field `nal_ref_idc` is set to the binary value `11` whenever the $nal_unit_type has the value `5`, `7` or `8`. When in packaging mode (i.e., packaging a fragment, as opposed to unpacking a fragment) the $nal_unit_type is assigned during the fragmenting phase (see Listing 3.2), while when in unpacking mode the same variable is assigned when the `rtp_nal_unit_type` field is parsed (see line 28 of Listing 3.4).

The multiplexing phase assures time-division multiplexing for a set of given fragments or frames of multiple audio, video or data tracks. Depending on the format, the multiplexing is done at a frame level or at a frame-fragment level. In order to achieve multiplexing at frame level, phase one of the adaptation is skipped. This phase outputs only at a given time or data limit. Such a limit is necessary to be able to produce media fragments of specified duration or size. The split is always done such that no reference between frames is lost.

The message layer packaging transformation adds extra headers or packets, such that the resulting fragment is recognised as valid by standard client protocol implementations. The syntax and semantics of this part of the DSL is rather similar to the *fragment layer packaging* part.

Many existing media container formats also contain a number of specific fields which are particularly hard to model. One example is the MPEG2 Transport Stream [93], which requires a 32-bit cyclic redundancy check value to be added to the Program Association Table package. In such a case we offer the possibility to add function "hooks" inside the DSL media container description. The compiler uses these to generate function templates, that developers can later implement.

3.3.3 Assuring the validation of timing constraints

Timing is an important dimension for assuring interoperability between streaming protocols. Message timing is not currently addressed in the CONNECT Mediator model. This is because most protocols have loose requirements with respect to message timing, with the sole role of assuring that opened network connections (which consume resources such as processing time and memory) do not persist indefinitely. Such enforced time-out events (e.g., the default connection timeout for persistent connections of Apache 2.0 httpd server

```
<Packaging trackid="1" type="video">
 <Packet name="rtp_video">
  <Rule type="select_fragment()">any</Rule>
  ...
  <Field name="SequenceNumber" value="$sequence_number">
   <InputRange startbit="16" bitlength="16"/>
  </Field>
  <Field name="Timestamp" value="$timestamp_90KHz">
   <!-- Sampling timestamp of the content. A 90 kHz clock rate MUST be used. -->
   <InputRange startbit="32" bitlength="32"/>
  </Field>
  ...
  <Field name="Payload" value="$track_id">
   <!-- RTP Payload Format for H.264 Video. See RFC6184 -->
   <InputRange startbit="96" bitlength="+"/>
   <Field name="forbidden_zero_bit" value="0b">
    <InputRange startbit="0" bitlength="1"/>
   </Field>
   <Field name="nal_ref_idc" value="10b">
    <InputRange startbit="1" bitlength="2"/>
   </Field>
   <Field name="nal_ref_idc" value="11b">
    <!-- a keyframe, a sequence parameter set or a picture parameter set -->
    <Rule type="select_fragment()">$nal_unit_type == 5 ||
     $nal_unit_type == 7 || $nal_unit_type == 8</Rule>
    <InputRange startbit="1" bitlength="2"/>
   </Field>
   <Field name="rtp_nal_unit_type" value="$nal_unit_type">
    <!-- Single NAL unit packet -->
    <InputRange startbit="3" bitlength="5"/>
   </Field>
   <Field name="rtp_nal_unit_type" value="28">
    <!-- Fragmented NAL unit packet -->
    <Rule type="select_fragment()">fragmented(2)</Rule>
    <InputRange startbit="3" bitlength="5"/>
   </Field>
   <Field name="FuHeader" value="28" optional="true">
    <Rule type="select_fragment()">fragmented(2)</Rule>
    <InputRange startbit="8" bitlength="8"/>
    <Field name="FragmentStart" value="$fragment_first(2)">
    </Field>
    <Field name="FragmentEnd" value="$fragment_last(2)">
    </Field>
    <Field name="ReservedBit" value="0b">
    </Field>
   </Field>
   <Field name="FragmentPayload" value="$fragment_data"/>
  </Field>
 </Packet>
</Packaging>
```

Listing 3.4 – RTP Container Format. Packaging phase for the H264 codec. Parts of the description are omitted. The full description can be found in the Appendix B.4

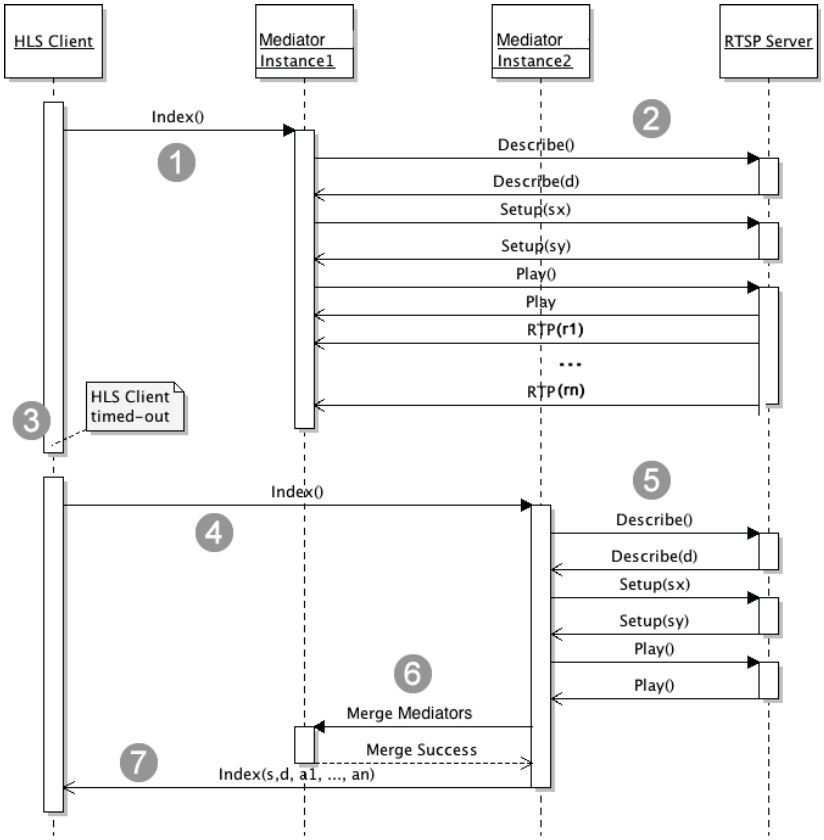


Figure 3.8 – Sequence diagram illustrating the merge of two RTSP-to-HLS Mediators

is 15 seconds [11]) do not necessarily pose an obstacle to interoperability.

On the contrary, real-time streaming protocols impose strict requirements on packet arrival and also on the inter-arrival variation (or jitter) as a means to assure the quality of service. This is why any processing done in-the-middle by a mediator should treat such requirements explicitly. To assure real-time streaming, multimedia packets (Figure 3.2, Elementary Stream) must be processed with respect to timing. First, packets which arrive late or are delayed during processing should be dropped at any phase by the mediator. This QoS policy takes place inside the *Media Parser* and *Media Composer*.

In the RTSP-to-HLS example, mediation is done between a real-time protocol and a non real-time protocol. This application driven heterogeneity leads us to the second timing related challenge of streaming protocols, which we call *application buffering requirements*. Buffering techniques differ depending on the streaming protocol either being done on the

[11] http://httpd.apache.org/docs/2.0/mod/core.html#keepalivetimeout

client side, on the server side or both. This is why buffering requirements of each protocol should be managed by the mediator. In the RTSP-to-HLS example, we see that the first request of the HLS client *!Index()* (Figure 3.5) triggers an *?Index(s, d,* $a_1, a_2, .., a_n$*)* response from the server. The fields of this message are: the sequence number s, the duration d in seconds of each chunk, and a list of stream chunk URLs $a_1, a_2, .., a_n$. The mediator is free to choose any chunk duration, but according to the HLS specification [88] "the client SHOULD NOT choose a segment which starts less than three target durations from the end of the Playlist file". In other words, at least three chunks should have already been cached by the server (in our case by the mediator) by the time of the *Index* request. Since the RTSP server does not buffer data, the mediator must assure it, but in this case, delaying the HLS *Index(s, d,* $a_1, a_2, .., a_n$*)* response by three chuck durations will exceed the HTTP response time-out of the client application.

Based on the CONNECT Mediator definition the two presented networked systems are semantically equivalent, that is, there exists a mapping to merge their respective colored LTS (in Figures 3.3 and 3.4) into a k-colored automaton (represented in Figure 3.5). However, the Mediator will fail at run-time because the *?Index()* request (in Figure 3.5) will trigger a time-out. This problem was anticipated in Section 4.4.3 of [53] where, in the case of the Bonjour-to-SLP experiment, mediation presents a 600 percent increase in response-time while still being low enough not to trigger time-out.

We propose a solution, for solving streaming application heterogeneity with respect to buffering requirements, based on the principle of locality. The principle of locality is widely used in many areas of computer science for a number of optimizations of systems, like: caching, pipelining, instruction prefetch, etc. If buffering (or in the more general case, a long operation) done by a Mediator triggers a time-out event on one side, the Mediator should be kept active with the other NS and not take any transition involving the (disconnected) NS. We first employ the principle of temporal locality. If, shortly after a first session triggering a response time-out, a second mediation session is initiated between two NSs, and the systems reach the same protocol state (that previously triggered the time-out), the two Mediator instances can be merged. By doing so, the time-out will not occur in the second session, because the first Mediator instance was able to complete the long operation (multimedia buffering in our case) during the elapsed time. Of course, this solution also assumes branch locality, that is, the second session between the two NSs will follow the same transition sequence as the first one. We conclude that reaching an equivalent state is necessary to merge two Mediator instances.

We illustrate the presented solution for the case of the RTSP-HLS example in the form of a sequence diagram (in Figure 3.8). The flow of messages used in the sequence diagram is given by the merged k-coloured automaton in Figure 3.5. At phase 1 the HLS client

Device	Samsung GT-I9000	Google Nexus One	iPhone 3G
Role	Server	Client	Client
Platform	Android 2.2.1	Android 2.3.4	iOS 4.2.1
CPU	1 GHz (S5PC110)	1 GHz (QSD8250)	412 MHz
Memory	512 MB	512 MB	128 MB
Media framework	PV OpenCORE	Stagefright	AV Foundation
Stream support	RTSP	RTSP/HLS	HLS

Table 3.2 – Mobile devices used in the experiment to assess AmbiStream's performance.

sends an *(*Index()) request to the Mediator. Next, in 2, the Mediator opens a connection to the RTSP server and initiates a streaming session by the sequence of request-response messages: *Describe*, *Setup*, *Play*. At this point, the RTSP server will start sending RTP messages, that will be buffered by the mediator in order to meet the requirements of the HLS client in terms of duration of the initial stream "chunk". Because the buffering period is greater than the response time-out enforced by the client, the HLS client will disconnect at step 3. We assume that the HLS client will retry to establish the connection by making an identical request *Index()* (marked as point 4). This request is treated by a second Mediator instance, which follows the same transition path, and eventually arriving in the same system state that triggered the time-out of the client. At this point 6, the Mediator should verify that merging is possible (i.e., the states are indeed equivalent). Because the stream "chunks" $a_1, a_2, .., a_n$ were pre-buffered by the first Mediator instance, at point 7 the *Index(s, d,* $a_1, a_2, .., a_n$*)* response is delivered immediately and not triggering time-out.

AmbiStream was modelled taking into consideration the architecture of modern smartphone platforms, such that resource critical operations (e.g., multimedia decoding) are managed by each platform internally. We prove that automated streaming protocol adaptation can be done locally on mobile phone platforms without sacrificing performance or extensibility. Furthermore, we enable legacy devices to employ unsupported streaming protocols by using an AmbiStream-enabled device as mediator intermediary. In order to evaluate the presented solution, we have implemented AmbiStream in Java and Objective-C and used it on AndroidOS and iOS.

3.4 Implementation and experimental results

The goal of the experiments presented here is to evaluate the overall performance of the AmbiStream middleware and the achievable stream quality while performing data adaptation on mobile devices. The experiments were performed on both Android and iPhone smartphones.

In both of the experiments presented below, the same set of source media files was used.

The test files have a duration of 210 seconds, are encoded with a single (H.264-avc video) track, have a CIF frame-size (352 by 288), and a frame-rate of 30 fps. The test is conducted for 16 different bit-rates between 50kbps and 1500kbps using the mentioned file format and content. Each set of tests is repeated at least three times, so each of the metrics presented is characterized by 168 minutes of video streaming to each client device. In total, more than 16 hours of streaming between smartphones were necessary. The mobile phones used are mentioned in Table 3.2.

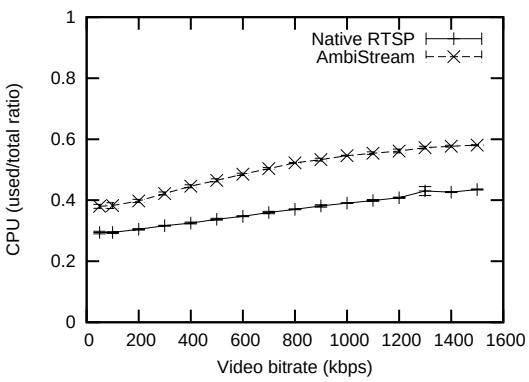
3.4.1 Collecting mobile device performance data

We have chosen to favour system-wide metrics to more specific ones (i.e., metrics of the application process) because we also make use of native system services and because mobile platforms do not frequently provide equivalent metrics. We use as metrics for device performance: the total CPU utilization and the system-wide used RAM memory. Quality of service metrics considered are the packet delay variation (also referred to as inter-arrival jitter, described in [87]) and packet loss ratio. The quality metrics are only provided for the case where the protocol is adapted. The values are obtained at the middleware level and should indicate the maximum bit-rate achievable while still providing satisfactory quality. The reference test cases, used to compare the overall performance, make use of system media services directly.

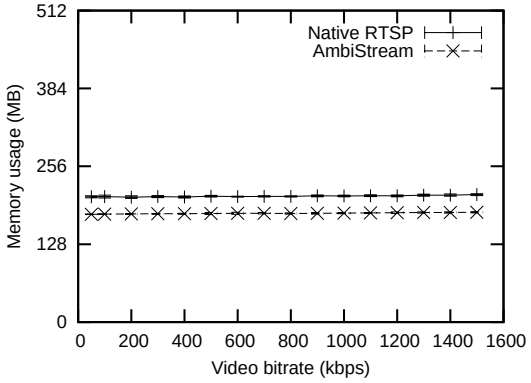
On Android mobile phones, the CPU and memory information is obtained by accessing the *proc* filesystem, used as an interface to the operating system kernel on most Linux based distributions. The logs are stored in the internal memory of both Android phones. To avoid that the access to the filesystem and data parsing are influencing the final results, the access to the */proc/stat* and */proc/meminfo* is done every five seconds, and the same file-descriptors are reused multiple times until the end of the test. On the iOS platform, system performance information was collected using the tools integrated with the development kit.

3.4.2 RTSP to HLS between Android and iOS smartphones

The second experiment consists of translating the RTSP protocol to HTTP Live Streaming, using two different client platforms: AndroidOS 2.3.4 and iOS 4.2.1. The choice of the smartphones is motivated by their native support of HLS. This way we can reason about the overhead introduced by our middleware layer with two different devices. Contrary to the first experiment, this one requires data conversion between RTP and MPEG-TS. MPEG-TS is one of the most used multimedia formats, most notably for digital television. The conversion from RTP to MPEG-TS requires a large number of transformations, thus



(a) CPU usage on Nexus One



(b) RAM usage on Nexus One

Figure 3.9 – AmbiStream performance on Nexus One (RTSP)

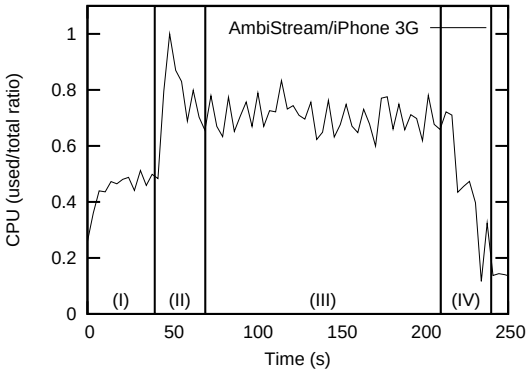


Figure 3.10 – Data capture (HLS)

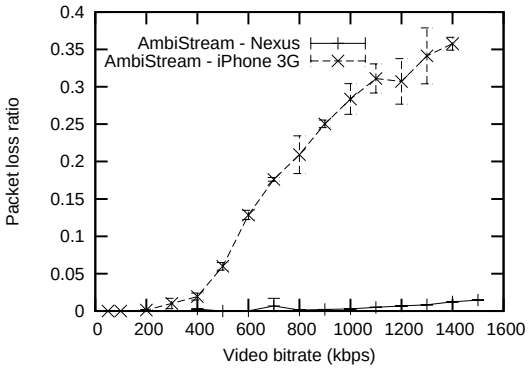
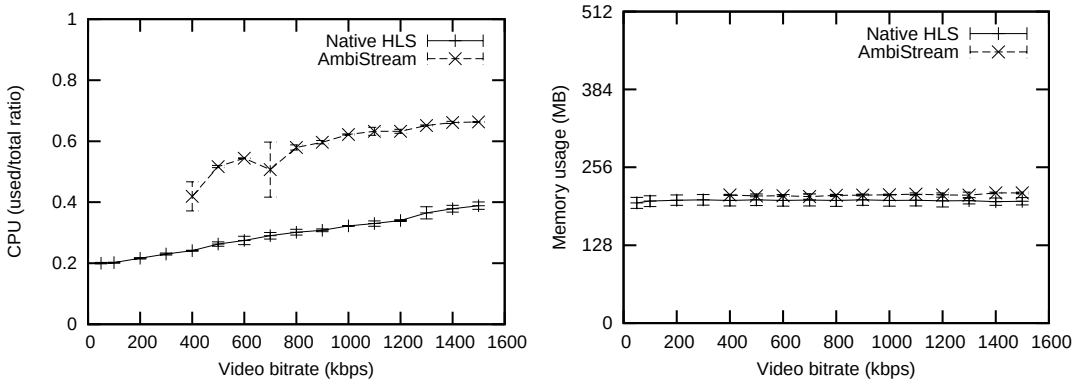


Figure 3.11 – Packet loss (HLS)

providing a good impression of achievable on-the-fly conversion limits of media formats on current generation smartphones.

Because HLS protocol requires the existence of a cached amount of content on the server-side before a client can connect (and begin playback), while the RTSP protocol does not, a 30s start-up delay is introduced by the middleware layer to allow protocol translation. During this period, less memory and CPU are used. To better evaluate the performance of the devices, we divide the experiment run in four periods (e.g., as shown in Figure 3.10 for CPU utilisation): *(I) the buffering period* (only multimedia data adaptation is performed), *(II) the media-player start-up* (causes a short increase in CPU usage), *(III) the streaming period* (both data adaptation and playback are performed) and *(IV) the stream-end* (the source has finished streaming, but the playback is continued until buffer depletion). Thus, only the part *(III)* of the observation was used to produce the results presented in Figures



(a) CPU usage (b) RAM usage

Figure 3.12 – AmbiStream performance on Nexus One (HLS)

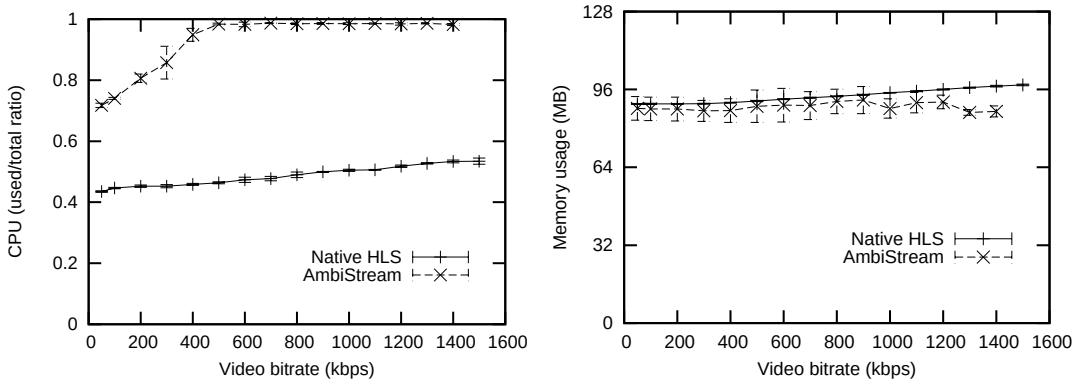
3.12 and 3.13.

As expected, the difference in container formats (RTP and MPEG-TS), increases the overhead of AmbiStream. For Android platform, the tests for bit-rates inferior to 400kbps (in Figures 3.12a and 3.12b) were discarded due to the existence of a minimal caching size, requiring a longer start-up delay. While on the Nexus One, the overhead introduced does not reach a quality limit for bit-rates below 1500kbps, the iPhone 3G is only able to adapt streams of up to 400kbps. Above this limit, the packet loss (see Figure 3.11) becomes noticeable and the media-player suffers playback stalls. The results on the iPhone are worse due to the significantly lower processing power and memory (see Figure 3.2). Nevertheless, according to the mobile platform providers, a 400kbps video bit-rate is considered to be medium/high quality for smartphones[12] [13]. Considering the results in Figure 3.13b, we see that the memory usage is decreasing (in the case of AmbiStream) for higher video bit-rates. This behaviour is normal considering the packet loss (see Figure 3.11).

3.5 Discussion

In this chapter we have identified the challenges raised by the heterogeneity of the streaming protocols of existing mobile phone platforms. Further, we have introduced the AmbiStream multimedia-oriented middleware architecture, designed to enable the multi-platform and multi-protocol interoperability of streaming services. We motivated our design choices with respect to the CONNECTor architecture using a concrete live streaming interoperability

[12] `https://developer.apple.com/library/ios/technotes/tn2224/_index.html`
[13] `http://developer.android.com/guide/appendix/media-formats.html`



(a) CPU usage (b) RAM usage

Figure 3.13 – AmbiStream performance on iPhone 3G (HLS)

example. Finally, we validated the applicability of the AmbiStream CONNECTor solution by a series of experiments on two mobile platforms.

The AmbiStream live streaming interoperability solution was designed to function in fully-distributed environments. This characteristic is vital for the deployment of networked systems in remote places and a priori unknown environments. A good example of such a circumstance is presented in [94] as The Joint Forest-Fire Operation, where an number of systems of key importance (such as IP cameras and UAVs) are enabled with multimedia streaming capabilities.

Unlike other data adaptation approaches, AnbiStream uses a two layer description of message formats. We recall that one layer corresponds to messages exchanged by the "control" part of the protocol, while the other corresponds to the multimedia data formats. While this approach is efficient for multimedia streaming protocols it is not applicable to other classes of multilayered data formats. For this reason, in what follows, we investigate the mechanisms by which pre-compiled (third-party) message translators can be reused and combined. As opposed to DSL-based approaches in general, the composition of pre-compiled message translators presents two important benefits. First, the reuse of pre-existing translators for standard message formats relieves developers of the task of hand-coding them directly or using a DSL. Secondly, this approach does not impose any restrictions on the number of message encapsulation layers, and it is agnostic to the domain of the protocol. The later argument does not invalidate in any way the AmbiStream approach, because message translators still have to be hand-coded whenever the format is not standardised or when an implementation is unavailable. In such a case the AmbiStream approach represents an efficient means of enabling data adaptation for technically incom-

patible streaming protocols.

Chapter 4

Reusing pre-compiled message translators

Contents

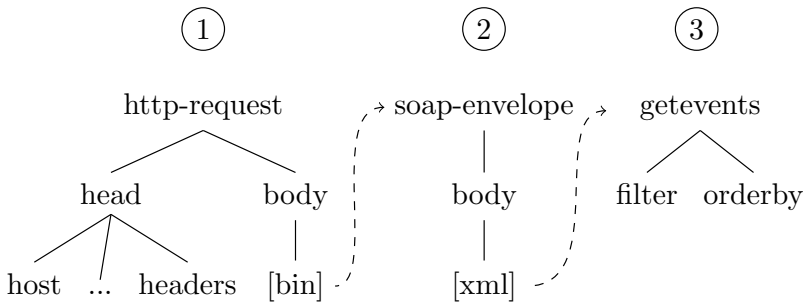
Message translation is challenged by the encapsulation of data according to different middleware protocols, e.g., SOAP message encapsulated within HTTP for Web Services. As a result, implementing message translators requires dealing with multiple message formats and identifying the parts of the message 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.

As an example, consider the message depicted in Figure 4.1a. The message combines two distinct data representations: ① a text-based message part that corresponds to the HTTP protocol, and parts ② & ③ that use XML serialization. In the case where individual translators for each independent message format are available (e.g., for parts ①, ② and

```
POST /api/default.asmx HTTP/1.1                                          1
Host: www.regonline.com
Content-Type: text/xml
Content-Length: 340
SOAPAction: "http://www.regonline.com/api/GetEvents”
APIToken: CC0TRrU5mhws9IRZIECHiMuahA+OZuaxuV

<?xml version="1.0" encoding="utf-8"?>                                   2
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/”>
   <s:Body>
      <GetEvents xmlns="http://www.regonline.com/api”>                   3
         <filter>Title.Contains(”ConnectTest")</filter>
         <orderBy>Title ASC</orderBy>
      </GetEvents>
   </s:Body>
</s:Envelope>
```

(a) Message sample



(b) AST

Figure 4.1 – A composed message sample and its associated AST

③), “glue code” has to be provided in order to compose them. However, to the best of our knowledge, existing methods for the composition of parsers are highly specific to a given parsing method or algorithm (e.g., grammar composition [95], parser-combinators [96], parse-table composition [76]) and cannot be easily generalized. As a result, existing methods do not allow for the systematic composition of message translators out of third-party translators, thereby requiring developers to implement hardcoded adapters in order to process messages.

The problem of composing message translators is related to the composition of the associated data structures (or inference of composite data types). The composition of data structures relative to the composition of message translators ensures that composite translators can be seamlessly (or even automatically) integrated with existing systems.

AST composition can be arbitrary, but it should not result in the loss of information. In other words, the AST transformation applied for the composition must be injective, thus invertible. For instance, when parsing the message in Figure 4.1a using a composite translator we would normally obtain three separate data structures, as shown in Figure 4.1b in the form of Abstract Syntax Trees (AST).

In the context of this work, we are particularly interested in the *substitution* class of AST transformations, that is, the substitution of a leaf node by a sub-tree. This allows us to represent encapsulated message formats in a hierarchical manner. In Figure 4.1b, we exemplify such a case of substitution using dotted arrows, as follows: (i) the node labeled [*bin*] in AST ① is substituted by AST ②, and (ii) the node labeled [*xml*] in AST ② is substituted by AST ③. While other compositions are possible (e.g., AST ② could alternatively be appended to the root of AST ①), this particular one closely resembles the way most protocols arrange encapsulated data, therefore being the most intuitive. The tree transformation mentioned above can be easily expressed by adapting already existing mechanisms for XML, such as the XSLT transformation language in combination with the XPath query language. In general, defining the composition of ASTs is rather straightforward. However, inferring the data structure resulting from an AST composition is more complex Indeed, it is already known that for an arbitrary tree transformation, the problem of type inference may not have a solution [7]. While the problem of type inference is quite common in the domains of functional programming languages [9, 10] and XML technologies [7, 8, 97], we are not aware of any solution capable of type inference for the *substitution* class of tree compositions although this kind of transformation is very common in practice (most notably in the XML transformation language XSLT).

The contribution of this Chapter is twofold:

1. Starting from the premise that "off-the-shelf" message translators for individual protocols are readily available in at least an executable form, we propose a solution for the automated composition of message translators. The solution simply requires the specification of a composition rule that is expressed using a subset of the navigational core of the W3C XML query language XPath [98].

2. Then, we provide a formal mechanism, using tree automata, that generates an associated AST *data-schema* for an arbitrary translator composition. This contribution enables the inference of correct data-schemas, relieving developers from the time-consuming task of defining them. On a more general note, the provided method solves the type inference problem for the *substitution* class of tree compositions in linear time on the size of the output. The provided inference algorithm can thus be adapted to a number of applications beyond the scope of this work, such as XML

Schema inference for XSLT transformations.

There is already a number of systems that can benefit from translator composition such as: Packet Analysers, Internet Traffic Monitoring, Vulnerability Discovery, Application Integration and Enterprise Service Bus, etc. As a result, our approach can have an immediate impact knowing that current implementations rely on tightly coupled and usually hardcoded message translators, to the detriment of software reuse.

The next section provides the background of our research focusing on challenges related to message parsing and translator composition Then, Section 4.1.2 details our approach to the systematic composition of message translators, while Section 4.2.2 introduces a method to automatically generate AST data-schemas (formalized as Hedge Automata) associated to a given composition of translators. Section 4.3 assesses our prototype implementation and its benefits. Finally, Section 4.4 summarises the contributions of this chapter.

4.1 Message parsing and composition

We remind that 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 Chapter 2 we discussed approaches that are used to implement or generate message translators. In what follows, we analyse existing approaches and discuss additional challenges regarding the composition of message translators. We focus on the problem of parsing, that is generally the most complex.

Parser composition is a difficult task for two reasons: (i) parsers are monolithic components that cannot be easily composed without regenerating them from the source grammar [76] (implying that they are generated from a context-free grammar or equivalent formalism, and not fully hand coded) and (ii) it is known that combining two arbitrary unambiguous grammars may result in an ambiguous output grammar, while the problem of ambiguity detection for context-free grammars is undecidable [77].

While parser generators allow the extensible or even incremental [99] 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.* [76,95] with a precise focus on *extensible programming languages*. A related approach, known as *combinatory parsing* [100], allows modular parser composition through a set of primitive operations. 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. The same can also be said about *combinatory parsing* approaches [100], since the building blocks (e.g., sequence, choice, repetition) of *parser combinators* map one-to-one to the constructions (i.e., derivation rules) of a CFG.

DSL and IDL-based parsers do not support composition and require messages to be defined in a monolithic way, which can easily become unmanageable for complex protocols.

All the aforementioned approaches are specific to a *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.

4.1.1 Composing heterogeneous message syntaxes

In a composite message format, ambiguity can occur between the outer (encapsulating) message format, called *host*, and (encapsulated) message format, called *extension*. Parsing ambiguity is known to be a theoretically hard problem [77] but in communication protocols, several solutions are commonly implemented to deal with it:

- *Context-aware parsing* [95,101] 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 errors. For example, the string `Hello <World>` can be transformed into `Hello &lt;World&gt;` to disambiguate it from XML markup syntax.

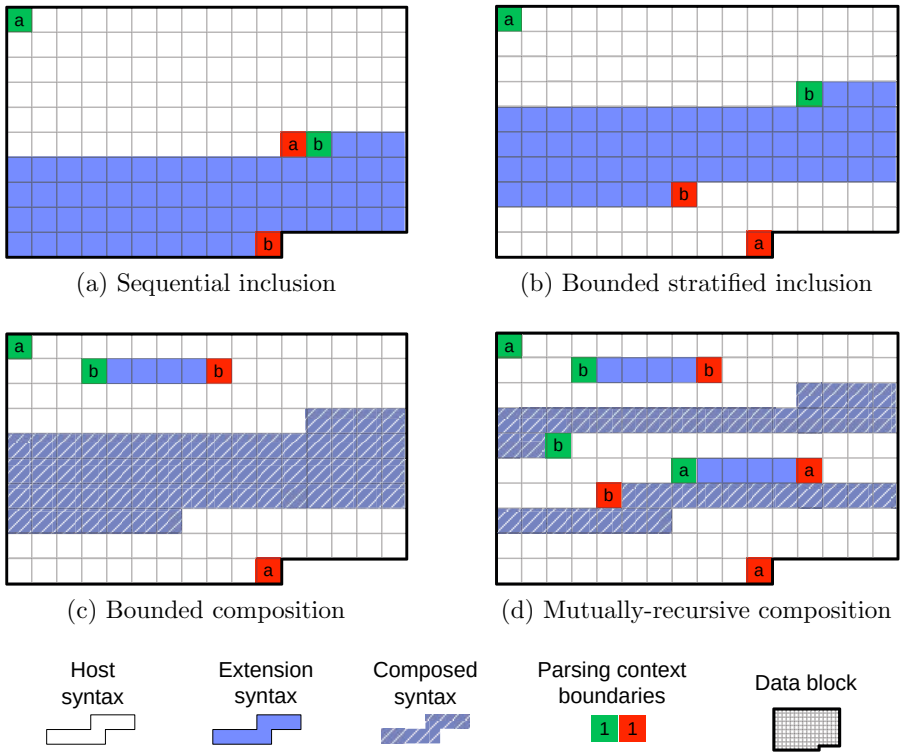


Figure 4.2 – Illustrative example of the four classes of syntax composition

- *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 4.2 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 4.2a, 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.* Commonly, 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 [102]. Figure 4.2b 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 the case above 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 4.2c) can be parsed neither by the host parser, nor by the extension parser. Another difference from the case above is that this class may include both *syntax extensions*, which allow expanding the initial language with new message types, as well as *syntax restrictions*, which introduce 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 [96, 100]), 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 [76, 95], where parser composition allows extending the syntax of a host programming language, for instance Java (e.g., context a a in Figure 4.2d), with an extension, 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. 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).

4.1.2 Message translator composition

Translators interpret network messages to produce a data structure that corresponds to the content found in the message. In the following sections, we model data structures as Abstract Syntax Trees (AST). The formal foundation of such trees is represented by *finite ordered trees*.

Definition 6 (Finite ordered tree) *A finite ordered tree t over a set of labels Σ is a mapping from a finite prefix-closed set of positions $\mathcal{P}os(t) \subseteq N^*$ into Σ. We denote by N the set of positive integers. We denote the set of finite strings over N by N^*.*

A finite (ordered) tree t over a set of labels Σ can also be defined as a partial function $t : N^* \to \Sigma$ with domain written $\mathcal{P}os(t)$ satisfying the following properties:

- *$\mathcal{P}os(t)$ is finite, nonempty, and prefix-closed.*
- *if $t(p) \in \Sigma$, then $\{j | pj \in \mathcal{P}os(t)\} = \{1, ..., k\}$ for some $k \geq 0$.*

Further properties regarding finite ordered trees are found in [103].

Definition 7 (Message translator) *A message translator comprises a parsing function $P : M \to T(\Sigma)$ that takes as input a bit-string and outputs a tree, and the inverse $C = P^{-1}$ where:*

- *$M \subseteq \{0, 1\}^*$*
- *$T(\Sigma)$ is a set of finite ordered, unranked, directed, and rooted trees labelled over the finite alphabet $\Sigma = \Sigma_0 \cup \{\beta, 0, 1\}$, where Σ_0 is a set of labels, not including the set of* binary labels *$\{0, 1\}$, neither the* binary-subtree label *β.*

Since elements of T represent ASTs, arbitrary data can be included only as leaf nodes, in the form of an ordered sequence of binary labels, by convention, under a β-labeled node. Such a structure (e.g., $\beta \to 1011...$) is equivalent to a bit-string $b \in \{0, 1\}^*$. We use this convention to avoid having an infinite label alphabet, such as $\Sigma_0 \cup \{0, 1\}^*$, that would be outside the scope of regular tree automata theory.

We detail our method of composing message translators in the form of two block diagrams, corresponding to the composition of, respectively, parser functions (in Figure 4.3) and the inverse composer functions (in Figure 4.4).

Informally, the parser composition method illustrated in Figure 4.3 works as follows. First, the stratified input message is parsed using the parser P_1, which corresponds to the first strata of the message. The user-provided query $\mathcal{Q}$ is then used to select positions in the resulting AST which correspond to encapsulated messages (of a second format). Then, the AST of the composite message is obtained by substituting in the initial tree every position that belongs to an answer to $\mathcal{Q}$ by trees resulting from the parsing of the encapsulated messages using P_2.

To better exemplify the translator composition mechanism, consider the stratified message presented in Figure 4.1 (the leaf nodes containing message data values are omitted). The first strata (or layer) ① of the message consists of a HTTP request message. By passing this message through a HTTP translator, we obtain an AST representation similar to Tree ①. Knowing that the sub-tree [*bin*] attached to the *body* node contains (encapsulates) SOAP message syntax we may pass this data to a SOAP translator to be interpreted. To support this kind of composition for all trees of the form of Tree ①, which may be an infinite set, we must generalize the composition mechanism. To do so, we can define the node-selection requirements as a *unary* (or *node-selecting*) *tree query* [97].

In the context of our work, we consider a tree-query to be a subset of the navigational core of the W3C XML query language XPath [98], which we represent formally in Section 4.2 as *tree query automata*. Using the XPath syntax, we can write $T_{HTTP}[/request/body/] \rightarrow T_{SOAP}$, meaning that the translator T_{HTTP} is composed with translator T_{SOAP} such that, for a given composite message, all nodes selected by the query $/request/body/$ are substituted with an AST corresponding to T_{SOAP}. While this example is trivial, more complex queries are supported. For instance, defining the composition between a HTTP translator and a MIME-type translator can be specified as $T_{HTTP}[/request/head/header[key =' Content - Type'] /value] \rightarrow T_{MIME}$, making use of an XPath predicate that enables the selection of a node that contains a specific value.

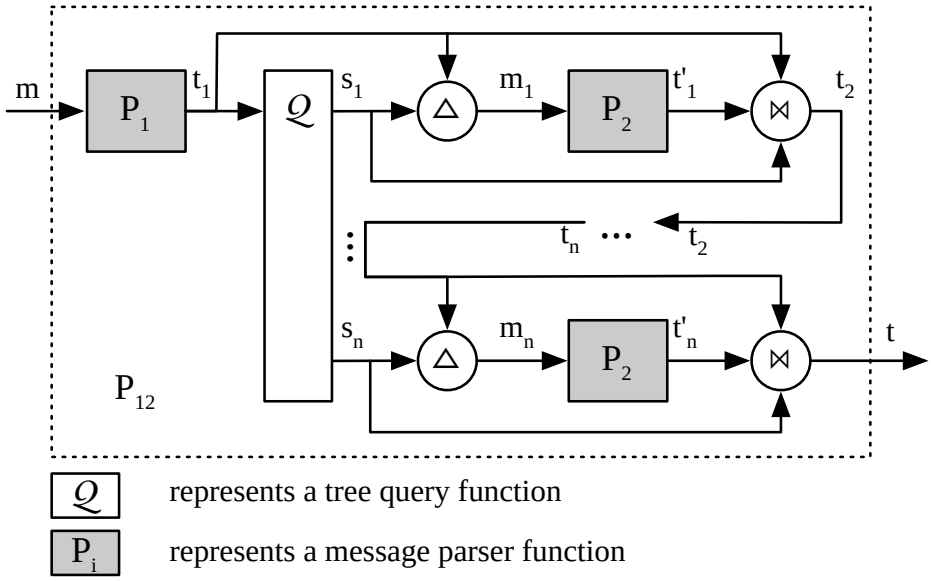


Figure 4.3 – Mechanism for composing parsers

Formally, we introduce the following definition for parser composition:

Definition 8 (Parser composition) *Given two message parsers $P_1 : M_1 \rightarrow T_1(\Sigma_1)$, $P_2 : M_2 \rightarrow T_2(\Sigma_2)$ and a user-defined tree query $\mathcal{Q}$ for $t_1 \in T_1(\Sigma_1)$, we define the composed parser $P_{12} : M_{12} \rightarrow T_{12}(\Sigma_1 \cup \Sigma_2)$ as follows (see Figure 4.3):*

- *For a stratified message $m \in M_1$, we apply the query $\mathcal{Q}$ on t_1, where $t_1 = P_1(m)$.*
- *The answer to $\mathcal{Q}$ for t_1 is $S = \{s_1, ..., s_n\}$, the set of selected positions in the tree t_1, with $n \geq 0$.*
- *For each $s_i \in S$, we compute $t_{i+1} = \bowtie (t_i, P_2(\triangle (t_i, s_i)), s_i)$ where:*
  - *$\triangle(t, s)$ denotes the selection of a bit-string from t at position s;*
  - *$\bowtie (t, t', s)$ denotes the replacement in t of a bit-string at position s by t'.*
- *The composed parser function $P_{12} : M_{12} \rightarrow T_{12}(\Sigma_1 \cup \Sigma_2)$, with $M_{12} \subseteq M_1$, is defined as $P_{12}(m) = t_{n+1}$ ($t_{n+1} = t$ in Figure 4.3).*

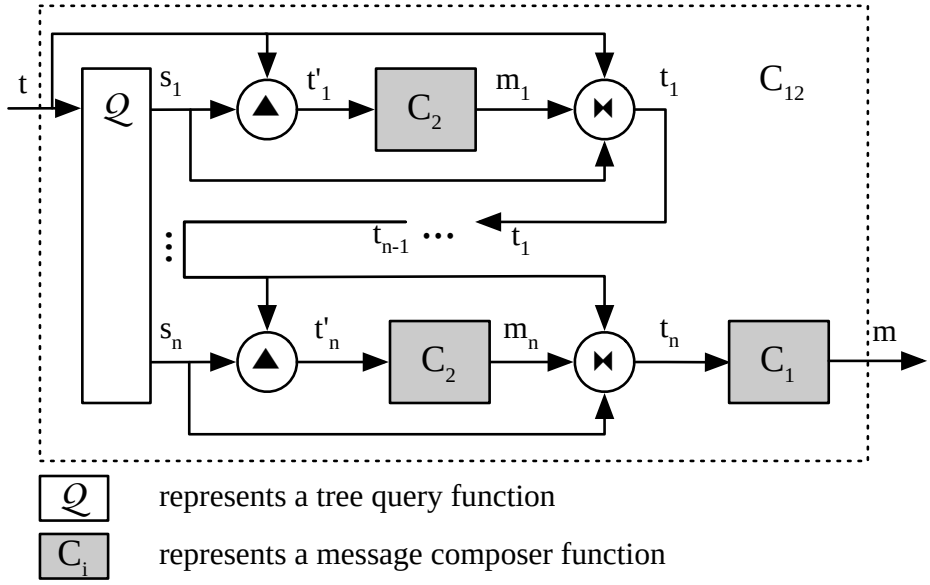


Figure 4.4 – Mechanism for composing un-parsers

Similarly, we have the following definition for un-parser (or composer) composition, which is illustrated in Figure 4.4:

Definition 9 (Composer composition) *Given two message composers* $C_1 : T_1(\Sigma_1) \rightarrow M_1$, $C_2 : T_2(\Sigma_2) \rightarrow M_2$ *and a user-defined tree query* $\mathcal{Q}$ *for* $t \in T_1(\Sigma_1)$, *we define the composed composer* $C_{12} : T_{12}(\Sigma_1 \cup \Sigma_2) \rightarrow M_{12}$ *as follows.*

- $C_{12}(t) = C_1(t_n)$, *where* $M_{12} \subseteq M_1$
- $t_0 = t$, $t_{i+1} = \bowtie (t_i, C_2(\blacktriangle(t_i, s_i)), s_i)$ *where:*
  - $\blacktriangle(t, s)$ *denotes the selection of a subtree from* t *at position* s
  - $\bowtie (t, m, s)$ *denotes the replacement in* t *of a subtree at position* s *by the bit-string* m.

In the compositions of both functions, we consider that the elements of the query result $S = \{s_1, ..., s_n\}$ are *prefix-disjoint*, meaning that for any position s_i of the form $s_i = s_j s'$, then $i = j$. This property ensures that the *selection* $\triangle(t, s)$ and *replacement* $\bowtie (t, t', s)$ operations for a query result S on a *tree* t can be performed in an arbitrary order.

We observe that the aforementioned composition method is part of a wider class of *result-conversion* mechanisms. Most notably, the Scala (`http://www.scala-lang.org/`) programming language implements a result-conversion *parser combinator*. A result-

conversion combinator, denoted $P\hat{}\hat{}f$, is defined as a higher-order function, which takes as input a parser function P and a user-defined function f that is applied on the result of P. The modified parser succeeds exactly when P succeeds. This method is particularly relevant to our case because it is purely defined on the output data type, and thus it does not require any knowledge about the input message syntax. However, in our case, the function f is not arbitrary, since it is represented by a user query.

By applying the composition mechanism defined above, we are able to compose message translators as black-box functions, which, in turn, allows the translation of composite messages (for the case of *stratified syntax inclusion*) to and from a uniform tree representation. However, the constraints on the structure of this tree representation (i.e., the data-schema) are unknown. In the next section, we provide a formal mechanism by which we are able to automatically generate a *data-schema* for the resulting ASTs.

4.2 Inferring the abstract data types of composite message translators

4.2.1 Background on type inference

As far as we know, the problem of inferring the output schema (or the data type) of an arbitrary tree transformation has not yet been solved, while it is known that, in general, a transformation might not be recognizable by a schema [7].

In [7], Milo *et al.* propose an approach capable of type inference for queries over semistructured data, and in particular XML. However, the query mechanism for node selection that is proposed is only capable of vertical navigation, meaning that the language does not allow conditions on the ordering of nodes (horizontal navigation), which is particularly required for selecting ordered nodes of an AST. In [8], Papakonstantinou *et al.* propose an improved approach that can infer Data Type Definitions (DTDs) for views of XML documents. In their work, *views* are in fact subtrees that are selected using a query language capable of both vertical and horizontal selection conditions. The solution can be easily generalized to select *views* from multiple trees/sources. However, it is not capable of merging the results from different XML languages, as it is required in the case of translator composition.

ℂDuce [9] is a language for expressing well-typed XML transformations. Specifically, ℂDuce can automatically infer non-recursive data types corresponding to a provided XML transformation. ℂDuce does not propose a high-level tree composition mechanism, but rather provides a language where XML queries and transformations can be implemented using a low-level form of navigational patterns that are non-compliant with the XPath

standard. In [10], an extension was provided, which essentially allows implementing XPath-like navigation expressions by pattern matching and to precisely type them. While this improves the query mechanism, it does not solve the limitations of the transformation mechanism which, in our understanding, is limited to disjoint trees concatenated in the result. We mention that the problem of type inference is related, but different to the problem of static type checking for XSLT transformations [97,104], which intends to verify that a program always converts valid source documents into also valid output documents for the case were both input and output schemas are known.

Considering the above, none of the approaches solve the AST type inference problem for the *substitution* class of transformations, which applies whenever two or more message translators are combined.

4.2.2 Data-schema composition

Data-schema languages share unranked tree automata as theoretical foundation [105]. In what follows, we use top-down non-deterministic finite hedge automata [106] (or, equivalently, hedge grammars) to model AST languages. We first recall below basic definitions associated with tree automata.

Definitions

Definition 10 (NFHA) *A top-down non-deterministic finite hedge automaton (NFHA) is a tuple $\mathcal{A} = (Q, \Sigma, Q_0, \Delta)$ where Q is a finite set of states, Σ is an alphabet of symbols, $Q_0 \subseteq Q$ is a subset of accepting states, and Δ is a finite set of transition rules of the type $q \to a(R)$ where $q \in Q$, $a \in \Sigma$, and $R \subseteq Q^*$ is a regular language over Q.*

Further, we introduce the definition of $\mathcal{A}$-derivations, which we use as a helper mechanism to describe the process of tree evaluation by a hedge automaton.

Definition 11 ($\mathcal{A}$-derivations) *Given an automaton $\mathcal{A}$, $\mathcal{A}$-derivations are defined inductively as follows. A tree $r \in T(Q)$ with $r = q(r_1 \dots r_n)$ is a derivation from a state $q \in Q$ for a tree $t \in T(\Sigma)$ with $t = a(t_1 \dots t_n)$ if:*

- *There exists a transition rule $q \to a(R) \in \Delta$ such that $q_1 \dots q_n \in R$,*
- *For all $1 \leq i \leq n$, r_i is an $\mathcal{A}$-derivation from $q_i \in Q$ for t_i.*

A tree $t \in T(\Sigma)$ is accepted by an automaton $\mathcal{A}$ if there exists a derivation from a state $q_0 \in Q_0$ for t.

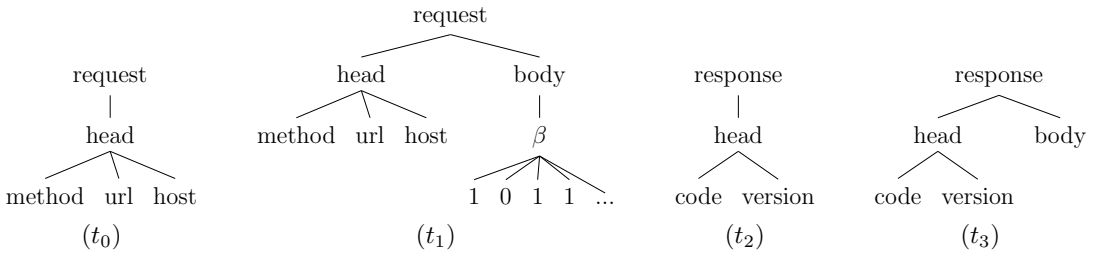


Figure 4.5 – Sample ASTs (leaf and β nodes omitted with the exception of the body part of t_1)

Example 1 (NFHA) Consider the automaton $\mathcal{A}^b = (Q^b, \Sigma^b, Q_0^b, \Delta^b)$ that recognizes the language $L(\mathcal{A}^b) = \{t_0, t_1, t_2, t_3\}$ containing the trees shown in Figure 4.5. The automaton is defined as follows:

— $Q^b = \{q_0, q_1, ..., q_{10}\}$, $Q_0^b = \{q_0, q_1\}$,

— $\Sigma^b = \{request, response, head, body, ...\} \cup \{\beta, 0, 1\}$,

— $\Delta^b = \{q_0 \to request(q_3\, q_2?), q_1 \to response(q_4\, q_2?), q_2 \to body(q_{10}),$
$q_3 \to head(q_5\, q_6\, q_7), q_4 \to head(q_8\, q_9), q_5 \to method(q_{10}), q_6 \to url(q_{10}),$
$q_7 \to host(q_{10}), q_8 \to code(q_{10}), q_9 \to version(q_{10}), q_{10} \to \beta(q_\beta)\},$
q_β is a state that accepts any sequence of $\{0, 1\}$ leaves.

The $\mathcal{A}_b$-derivation for the tree $t_0 = request\langle head\langle method\, url\, host\rangle\rangle$ (shown in Figure 4.5) is the tree $r = q_0\langle q_3\langle q_5\langle q_{10}\langle q_\beta\rangle\rangle\, q_6\langle q_{10}\langle q_\beta\rangle\rangle\, q_7\langle q_{10}\langle q_\beta\rangle\rangle\rangle\rangle$, where $r \in T(Q^b)$, thus, we can say that t_0 is accepted by $\mathcal{A}^b$.

While data-schemas are formalized as NFHA, we introduce a second type of tree automata, which we use to formalize tree queries. Informally, a query automaton is a tree automaton with the attachment of a set of "marked" states that are used to model node-selection.

Definition 12 (Query NFHA) *A query NFHA is a pair $\mathcal{Q} = (\mathcal{A}, Q_m)$ where $\mathcal{A}$ is a top-down NFHA over a set of states Q, and $Q_m \subseteq Q$ is a subset of marked states. Given a query $\mathcal{Q} = (\mathcal{A}, Q_m)$ and a tree t, a set S of positions in t is an answer to $\mathcal{Q}$ for t if there exists an $\mathcal{A}$-derivation r for t such that S is the set of positions of all nodes in r that are in $\mathcal{Q}_m$.*

Example 2 (Query NFHA) Consider the tree query $\mathcal{Q}^p = (\mathcal{A}^p, Q_m^p)$, which applied on trees from $L(\mathcal{A}^b)$, selects the node labeled *body* that is a child node of the tree root.

Intuitively, this query should return an empty set of positions S for the trees t_0 and t_2, and a single position when applied on t_1 and t_3. This query is defined as follows:

— $Q^p = \{q_0, q_1, q_2, q_\top\}$, $Q_0^p = \{q_0, q_1\}$, $\Sigma^p = \Sigma^b$, $Q_m^p = \{q_3\}$,

— $\Delta^p = \{q_0 \to request(q_\top, q_2), q_1 \to response(q_\top, q_2), q_2 \to body(q_3),$
$q_3 \to \beta(q_\beta)\}$, $q_\top$ is a state which accepts all trees.

Tree automata composition

We now present a method for composing two hedge automata $\mathcal{A}^b$ and $\mathcal{A}^e$ based on the composition rules defined using a query NFHA $\mathcal{Q}$. The resulting automaton $\mathcal{A}$ recognizes a tree language corresponding to the substitution defined by $\mathcal{Q}$.

Let $\mathcal{Q} = (\mathcal{A}^q, Q_m)$ be a query NFHA, where $\mathcal{A}^q = (Q^q, \Sigma^b, Q_0^q, \Delta^q)$.

Given two trees t^b (base tree) and t^e (extension tree), we note $t^b[\mathcal{Q} \leftarrow t^e]$ the tree obtained by substituting t^e in t^b at every position that belongs to an answer to $\mathcal{Q}$ for t^b.

Given two sets of trees T^b and T^e, we note $T^b[\mathcal{Q} \leftarrow T^e]$ the set of trees of the form $t^b[\mathcal{Q} \leftarrow t^e]$ where $t^b \in T^b$ and $t^e \in T^e$.

Since the composition performs the intersection between the base automaton and the query, we can suppose without loss of generality that the base automaton and the query share the same alphabet. Furthermore, it is worth noticing that the query can restrict the language recognized by the base automaton. However, in practice, we consider mostly queries that are expressed using XPath: such a query accepts all trees, even if the XPath query does not select any node in some of these trees.

```
1  path ::= '/' relative-path
2  relative-path ::= step[pred]
3                  | step[pred] '/' relative-path
4                  | step[pred] '//' relative-path
5  step ::= label
6         | '*'
7  pred ::= path | not(path)
8         | pred and pred
9         | pred or pred
10        | true
```

Listing 4.1 – Core XPath language used in data schema composition

We further consider the core XPath language given in Listing 4.1. We restrict `pred` to predicates that can be recognized by Boolean combinations of paths (with the usual set-to-Boolean interpretation: a path is true if and only if it matches at least one node).

This ensures that these predicates can be recognized by hedge automata. Indeed, the transformation from an XPath to a query automaton is straightforward, and it is done inductively over the structure of the path. The most relevant construction is `step[pred] '/' relative-path`: given the query automata A_P (with initial state Q_P) for `pred` and A_R for `relative-path`, a new accepting state q_0 is introduced with the transition $q_0 \to step(q_\top * q_P q_\top *)$ and the resulting automaton is intersected with A_R. Resulting automata are completed such that they accept all trees, even in the case that no node is selected.

For an arbitrary tree transformation, type inference may not have a solution [7]. It is thus important to prove that for the *substitution* class of tree transformations, which we presented in the beginning of this Chapter, all resulting AST languages are recognisable:

Proposition 1 *Given a query $\mathcal{Q}$ and two finite hedge automata:*
$\mathcal{A}^b = (Q^b, \Sigma^b, Q_0^b, \Delta^b)$ and $\mathcal{A}^e = (Q^e, \Sigma^e, Q_0^e, \Delta^e)$,
the language $L(\mathcal{A}^b)[\mathcal{Q} \leftarrow L(\mathcal{A}^e)]$ is recognisable by a finite hedge automaton $\mathcal{A}$.

Proof. It suffices to consider $\mathcal{A} = ((Q^b \times Q^q) \cup Q^e, \Sigma^b \cup \Sigma^e, Q_0^q, \Delta^e \cup \Delta)$, where Δ contains all the transitions rules of the form $(b, q) \to a(R)$ when:

— either $b \to a(R^b) \in \Delta^b$ and $q \to a(R^q) \in \Delta^q$ and R is the set $(q_1, q_1^q) \dots (q_n, q_n^q)$ when $q_1 \dots q_n \in R^b$ and $q_1^q \dots q_n^q \in R^q$.

— or $q \in Q_m$ and $b \to a'(q_f) \in \Delta^b$ and there exists $q_0 \in Q_0^e$ such that $q_0 \to a(R) \in \Delta^e$.

R is regular since R is recognized by the product of the automata that recognize respectively R^b and R^q. Based on this result, in Algorithm 1 we provide the procedure to generate $\mathcal{A}$. Next, we provide a proof that the algorithm is guaranteed to terminate with an answer for any valid input.

Proposition 2 *The Algorithm 1 terminates for all valid inputs.*

Proof. Let $\alpha \in \mathbb{N} \cup \{\omega\}$ be the number of loop iterations within the while loop between Lines 5 and 31 (possibly ω in case of non-termination). For every $i < \alpha$, let U_i be the value of $S \cup (Q^b \times Q^q) \setminus Q$ at Line 5 at the ith loop iteration. The loop satisfies $U_{i+1} \subsetneq U_i$ for every i such that $i + 1 < \alpha$. Therefore $(U_i)_{i<\alpha}$ is a sequence of strictly decreasing finite sets, thus, necessarily, $\alpha \neq \omega$.

Complexity. The size of the resulting automaton is $\mathbf{O}(|Q^e| + \left|Q^b\right| \times |Q^q|)$ and the running time is linear in the size of the output. The worst case is reached for a family of pairs of automata $(\mathcal{A}_i^b, \mathcal{Q}_i)_i$ where, for every pair of automata $(\mathcal{A}_i^b, \mathcal{Q}_i)$, every pair of states is reachable during synchronous exploration of $\mathcal{A}_i^b$ and $\mathcal{Q}_i$.

4.3 Assessment

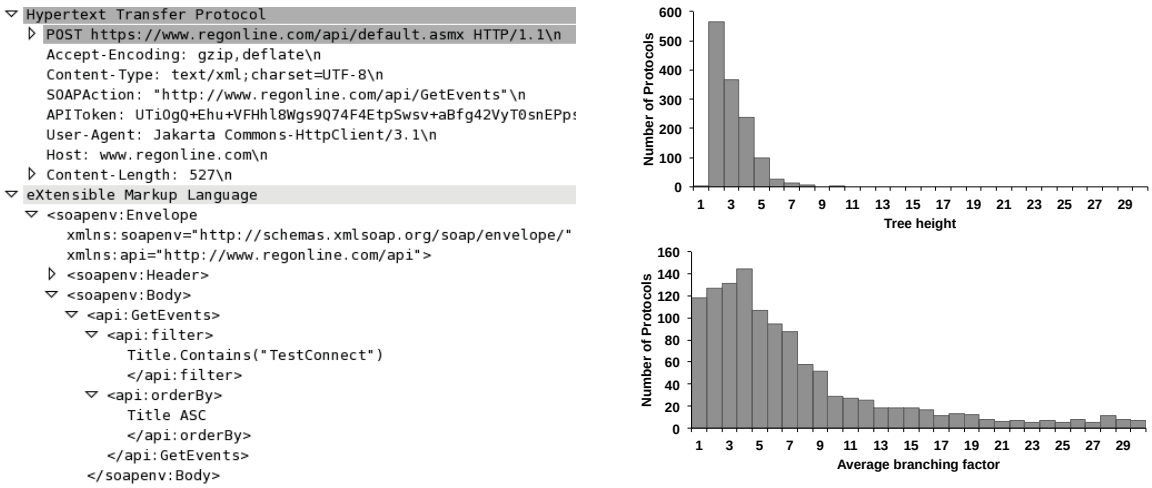
We have implemented a prototype of the proposed approach to the systematic composition of message translators, in the form of a Java library, which is available as open-source [1]. The library implements both the translator composition mechanism presented in Section 4.1.2, as well as the type inference algorithm introduced in Section 4.2.2. The purpose of this implementation is to be integrated in systems requiring high adaptability to new protocol stacks. Such systems include Enterprise Service Buses, Firewalls, Network Analyzers, etc.

While the underlying source code closely follows the formal mechanisms (such as tree automata) and algorithms presented in this chapter, we further concerned ourselves with making this library usable for non-expert developers by adhering to well-established standards. Specifically, the following abstractions are concretized, as follows: (i) AST types which are internally modeled as top-down NFHAs, are transformed both on the input and output to RelaxNG (`http://relaxng.org/`) or XSD (`www.w3.org/TR/xmlschema-1/`) schema documents, and (ii) AST query inputs, which we model as query NFHAs, are to be provided using a subset of the XPath query language (`www.w3.org/TR/xpath/`).

Translator support. A prerequisite of any composition is the existence of individual translators for each protocol. In our current implementation, we integrated translators for common middleware protocols like HTTP and SOAP, as well as generic translators for extensible formats such as XML and JSON. It is important to mention that while SOAP message formats are XML-based, they are more restrictive with respect to the messages accepted, and SOAP translators also produce more compact ASTs for the same messages. It is thus interesting to have protocol-specific translators, even when the protocol itself uses an extensible data format. Other translators may be easily integrated, although the creation of associated ASTs and AST data-schemas is currently hardcoded. To overcome this limitation, we are working on a solution that will automatically inspect third-party translators using reflection and generate the two according to the data-model used by the translators.

Translator composition in Wireshark. To assess the utility of our approach in a broader context that mediation, we discuss our contributions in relation to the well-known open-source packet analyzer software Wireshark (`http://www.wireshark.org/docs/dfref/`). The role of Wireshark is to capture network packets, to parse their content and to present the information to the user in a structured format for analysis. Figure 4.6a

[1] The project's Git repository can be checked out through anonymous access using a GIT client: `git clone https://gforge.inria.fr/git/iconnect/iconnect.git` (sourcecode located under the subproject `mtc`). Additionally, the repository can be browsed via the Git Repository Browser using the same URL. The `mtc` project is located under `projects/iconnect/iconnect.git/tree/mtc/`



(a) View of a message AST

(b) AST complexity

Figure 4.6 – Use of translators (aka packet dissectors) in Wireshark

depicts the representation of a HTTP/SOAP message in the Wireshark graphical user interface. Wireshark provides two mechanisms for composing translators (which they call dissectors). The first involves writing "glue-code" as an extension of an already existing dissector implemented in the C language. A more advanced solution (postdissectors and chained dissectors), which is similar to our composition approach uses the scripting language Lua (http://wiki.wireshark.org/Lua/Dissectors) to define compositions. However, postdissectors have to be implemented in Lua, thus eliminating the possibility of using already compiled, third-party, translators. Unlike the substitution-based composition that we introduce for ASTs, Wireshark uses a much simpler approach where disjoint trees are concatenated in the result. This can be observed in Figure 4.6a for the trees *Hypertext Transfer Protocol* and *eXtensible Markup Language*. In Wireshark, message ASTs do not have an associated data-schema, meaning that neither individual ASTs, nor ASTs resulting from a dissector chaining can be validated or inspected before run-time. Thus, the benefit of using our composition approach in a Packet Analysis software like Wireshark, enables: (i) composition/integration of third-party translators with already existing translators, and (ii) run-time validation and static-analysis/inspection of AST data types. The second is particularly beneficial when resulting data have to be further processed (e.g., data mining and machine learning) and stored (e.g., in a relational data base) rather than simply presented to the user.

Development effort. Our entire approach is based on the assumption that developers

are able to define XPath queries on message ASTs, in order to identify positions where data corresponding to composed protocols is located. In this case, it is important to estimate if this operation is effortless for developers in the general case. To this end, we conducted an analysis of AST type complexity on the 1317 protocols supported by Wireshark, focusing on two aspects that may influence the effort required to specify queries: tree height and branching factor. Intuitively, the tree height is proportional to the length of a query, when using only child axis (denoted `'/'`), and can prove complex to write in case of deep trees. Further, high branching factors (i.e., the average number of children at each node) also make queries more complex with respect to horizontal exploration, based on node order and sibling axis. As we show in Figure 4.6b, both parameters are rather low in general, with the most frequent tree height being of value 2, and the most frequent branching factor of 4. The parameters above are estimated based on the Display Filter Reference of Wireshark. Display Filters in Wireshark are quite similar to AST queries, although they are used for filtering network packets based on a predicate, rather than composing message translators. We note however, that this is only an empirical estimation knowing that the Wireshark Display Filter Reference only includes fields which are relevant for filtering, and that the hierarchical nature of message fields was deduced from the structure of field names. Furthermore, in the Display Filter Reference there is no notion of optional and mandatory fields. In the absence of this information, in the above, we considered that all fields are required. For this reason, we had to manually filter a small number of protocols which define an extensive number of optional fields (e.g., 1634 fields for the Financial Information eXchange Protocol –FIX–).

As a conclusion, we argue that the approach introduced in this chapter enables developers to design composite translators seamlessly as opposed to implementing hand-coded adapters. This statement is supported by the empirical evaluation above showing that, in the general case, the XPath queries that must be provided by the developers have a low complexity.

4.4 Discussion

In this chapter, we presented a method for composing message translators for complex protocols stacks by reusing already exiting translator components. For systems like Packet Analyzers, Firewalls, Enterprise Service Buses, etc., the reuse of third-party translators is critical since they must constantly evolve to support an increasing number of protocol stacks. The composition approach that we introduced in this chapter functions as a purely "black-box" mechanism, thus allowing the use of third-party parsers and message serializers independently of the parsing algorithm they use internally, or the method by which

they were implemented/generated. Our solution goes beyond the problem of translator composition by inferring AST data-schemas relative to translator compositions. This feature allows newly generated translators to be seamlessly (or even automatically) integrated with existing systems. On a more general note, the provided inference method solves the type inference problem for the substitution class of tree compositions. This contribution has a wider domain of applications beyond the specific scope of this work, such as the inference of XML schemas for XSLT transformations.

We implemented a prototype of the approach, which is released as open-source, to showcase its benefit in reducing development time by enabling seamless integration of message translators as reusable software components.

Algorithm 1 Tree automata composition

procedure COMPOSE($\mathcal{A}^b, \mathcal{A}^e, \mathcal{Q}$)
  $\mathcal{A} = (Q, \Sigma, Q_0, \Delta)$;
  $Q \leftarrow Q^e; \Sigma \leftarrow \Sigma^b \cup \Sigma^e; Q_0 \leftarrow Q_0^q; \Delta \leftarrow \Delta^e;\ S \leftarrow Q_0^b \times Q_0^q$
  while $S \neq \emptyset$ **do**
    $(b, q) \in S;\ S \leftarrow S \setminus \{(b, q)\}$
    if $q \notin Q_m^q$ **then**
      for all $b \rightarrow a(R^b) \in \Delta^b$ **do**
        for all $q \rightarrow a(R^q) \in \Delta^q$ **do**
          $R \leftarrow \{(q_1, q_1'), (q_2, q_2'), ..., (q_n, q_n') \mid \exists n, q_1...q_n \in R^b, q_1'...q_n' \in R^q\}$
          $\Delta \leftarrow \Delta \cup \{(b, q) \rightarrow a(R)\}$
          $S' \leftarrow \{(b', q') | (b', q') \text{ occurs in } R\}$
          $S \leftarrow S \cup (S' \setminus Q)$
          $Q \leftarrow Q \cup S'$
        end for
        if $q = q_\top$ **then**
          $\Delta \leftarrow \Delta \cup \{(b, q) \rightarrow a(R^b)\}$
          $S' \leftarrow \{(b', q_\top) | b' \text{ occurs in } R\}$
          $S \leftarrow S \cup (S' \setminus Q)$
          $Q \leftarrow Q \cup S'$
        end if
      end for
    else
      for all $b \rightarrow a(q_f) \in \Delta^b, q_f \in Q^b$ **do**
        for all $q_0 \in Q_0^e$ **do**
          $\Delta \leftarrow \Delta \cup \{(b, q) \rightarrow a(q_0)\}$
        end for
      end for
    end if
  end while
  return $\mathcal{A}$
end procedure

Chapter 5

A unified mediation framework for protocol interoperability

Contents

In this chapter, 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, such as the ones elaborated in CONNECT, in order to manage cross-layer data dependencies. More specifically, we make the following contributions:

- *Composite Cross-Layer (CCL) message translators.* We leverage the approach of Chapter 4 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 standard and 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., inferring schemas from XML documents).

- *A unified mediation framework.* [38] introduced 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 chapter is organised as follows. Section 5.1 introduces the interoperable conference management example, which we use throughout the chapter to motivate and illustrate our mediation approach. Section 5.2 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. The latter is the focus of Section 5.3, which details our approach to the generation of CCL translators by reusing and composing legacy ones. Section 5.4 describes a prototype implementation of the unified mediation framework and reports on the experiment we conducted using the interoperable conference management example. Finally, Section 5.5 concludes the chapter and discusses future work.

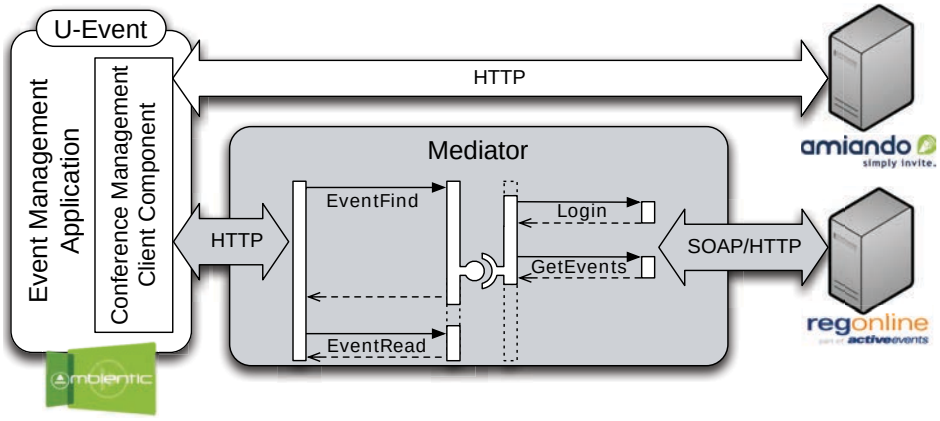


Figure 5.1 – Interoperability scenario

5.1 The interoperable conference management example

To motivate and illustrate our approach, we consider the Ambientic application for event management, called U-Event (see Figure 5.1). U-Event embeds a *client component* implemented as an Amiando client (`http://developers.amiando.com/`). U-Event needs 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 work 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 then 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 the information about a conference, clients issue an `EventRead` request with the event identifier as a parameter. To produce the equivalent result, Regonline clients 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 Re-

gonline 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

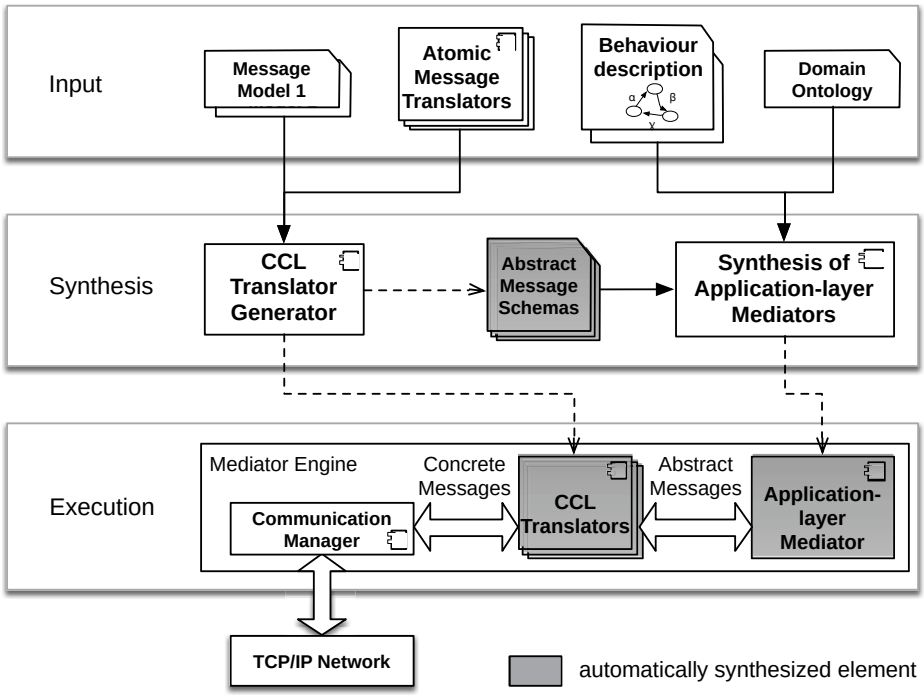


Figure 5.2 – A unified mediation framework

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.

5.2 Proposed 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 5.2 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 5.2 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*.

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 [42, 107–109]. To design this component, we closely follow the mechanism introduced by Bennaceur et al. [38]. In the following, we briefly describe the gist of their approach. To synthesise an application-layer mediator, a semantic correspondence between the messages sent by one component and those expected by the other component must be found. This task is known as interface

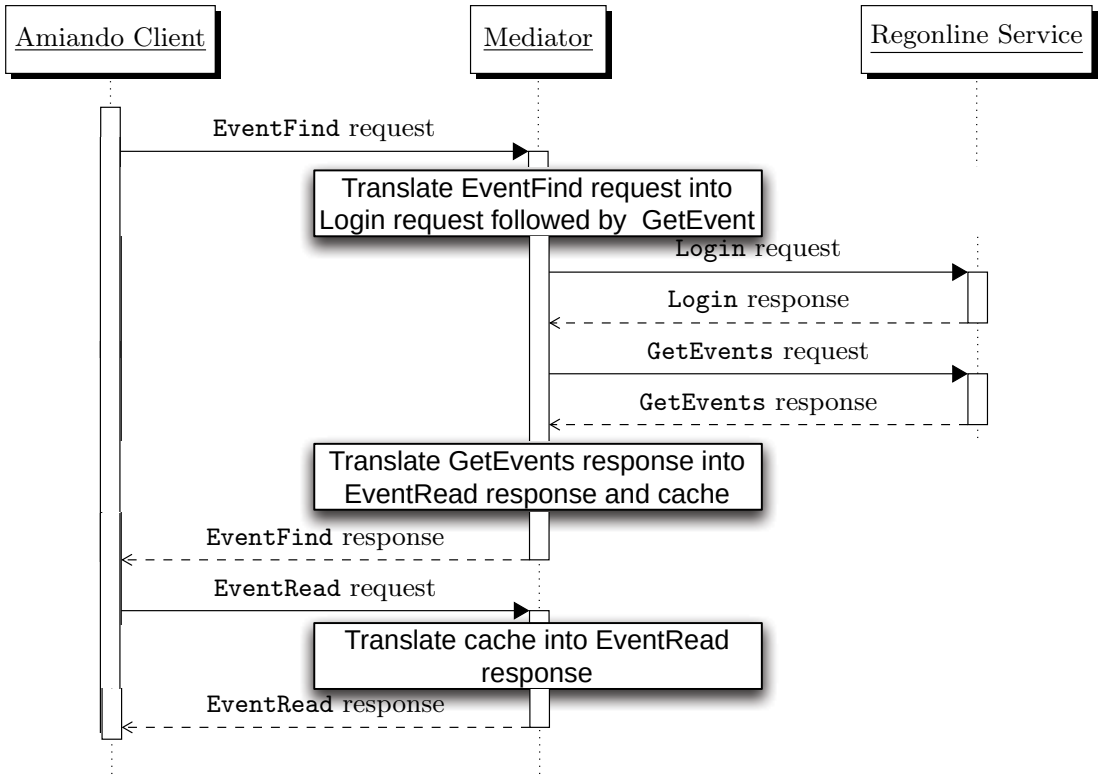


Figure 5.3 – Illustrating the mediator between an Amiando client and the Regonline service

matching. Interface matching is formulated as a constraint satisfaction problem and constraint programming is used to solve it. The approach further incorporates 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, an associated matching process is generated that performs the necessary data conversions between the actions of the components' interfaces.

Figure 5.3 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 5.2, 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 and TLS/SSL.

5.3 Cross-layer mediation

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

5.3.1 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 a set of XML documents. The same tool can be used to infer schemas for JSON and 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.

[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 tokens to produce the corresponding ASTs.

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`.

5.3.2 Composition of message translators

We mentioned that *Atomic message translators* are combined based on a *Message Model*. In Listing 5.1 we present a fragment of the *Message Model* describing the *Regonline service*. The full description can be found in the Appendix C.1. 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 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

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

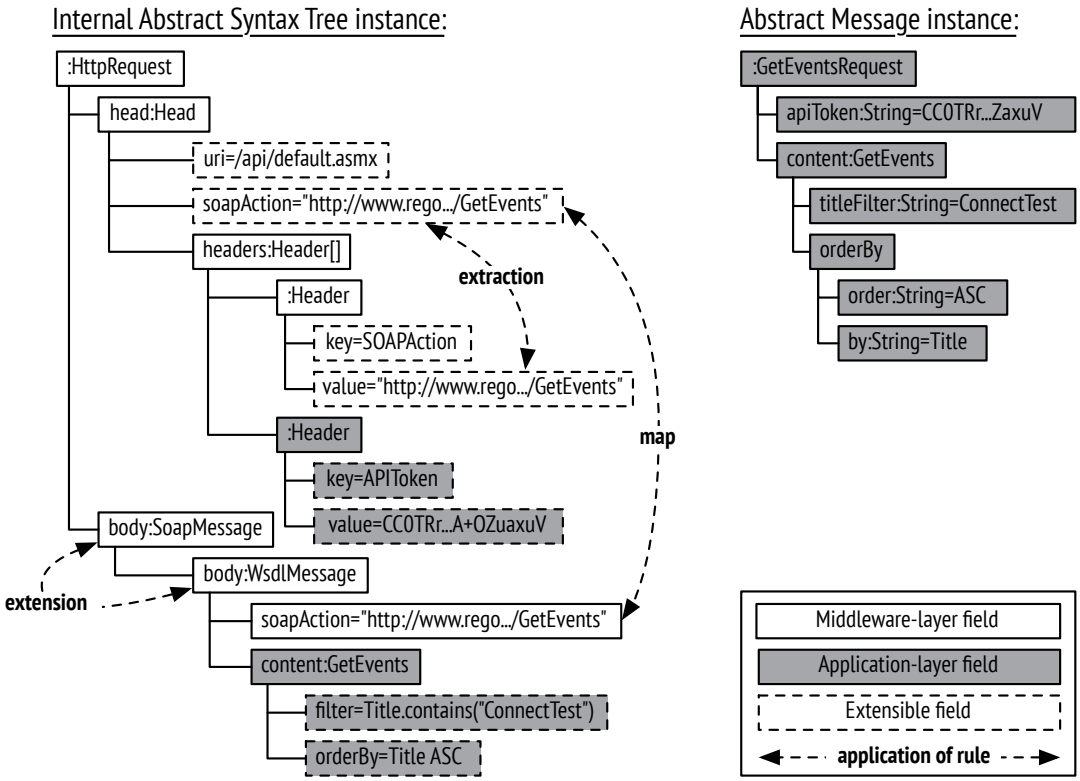


Figure 5.4 – AST of Regonline GetEvents-Request

message translators to produce *Internal AST instances*. As an illustration, a CCL AST instance of the Regonline `GetEvents Request` message is given in Figure 5.4. 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. We remind that in Chapter 4 we provided 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

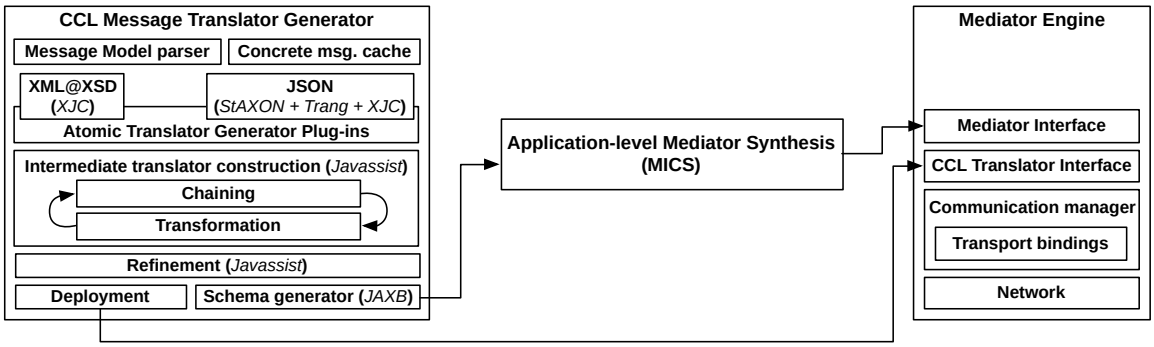


Figure 5.5 – Implementation of the unified mediation framework

mapping of data. Finally, we generate the functions necessary to transform *Abstract Messages* into their corresponding *Internal AST* representation, and the inverse.

5.3.3 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 presented in Section 4.2, 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 5.4) 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 rely on multiple middleware layers, message composition may require data to be mapped internally between layers. 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 5.4, 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.

5.4 Implementation and validation

We have implemented a standards-based prototype of the proposed mediation framework using Java, following the architecture described in Figure 5.5. 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 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 5.1) 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 for each system using the CCL translator generator tool (see Figure 5.8). Then, we provide two message samples containing the JSON formatted responses of the Amiando service. The CCL translator generator is then able to generate eight different composite message translators (listed in Figure 5.7) 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. Based on the SAXSDs, the provided do-

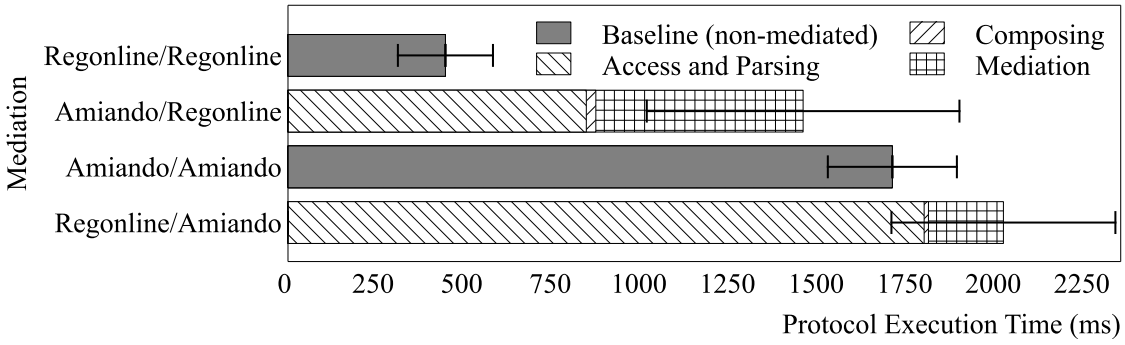


Figure 5.6 – Comparison between mediated and non-mediated executions

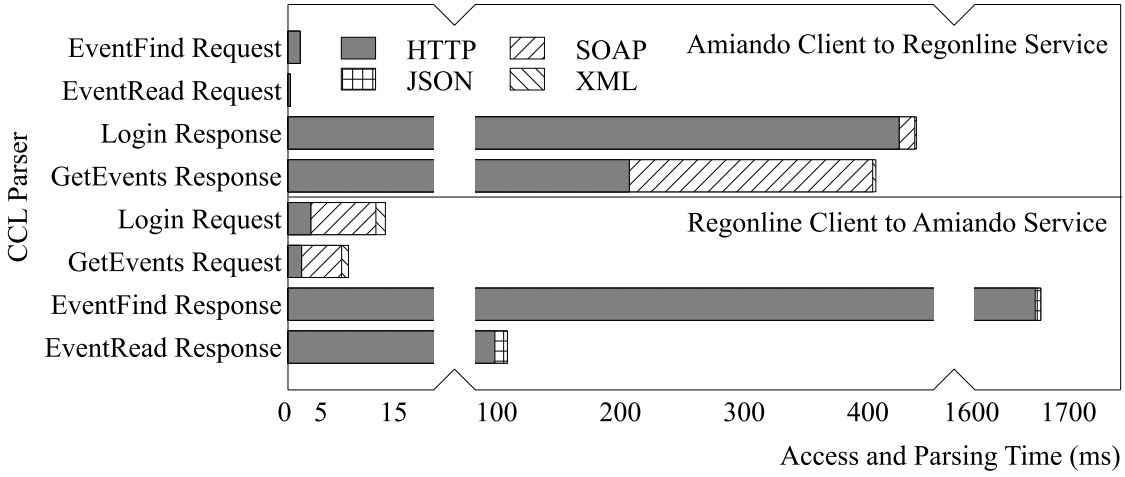


Figure 5.7 – Access & Parsing time decomposed by Atomic parsers

main ontology (fragment shown in Figure 5.9) and the LTS behavioural descriptions, MICS generates two mediators (Amiando server to Regonline client and the complementary. See Figure 5.10).

We compare the mediated execution-time with the non-mediated case. Each test was repeated 30 times, in similar conditions, and connection delays were excluded (e.g., opening sockets, SSL handshake, etc).

In Figure 5.6, 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 5.7, 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 5.6) can be reduced by using a more efficient SOAP Atomic parser. Comparing to the non-mediated tests, we can conclude that our mediation approach introduces an acceptable overhead while enabling seamless interoperability between two originally incompatible systems.

5.5 Discussion

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 chapter 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.

Open issues include 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, run-time 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.

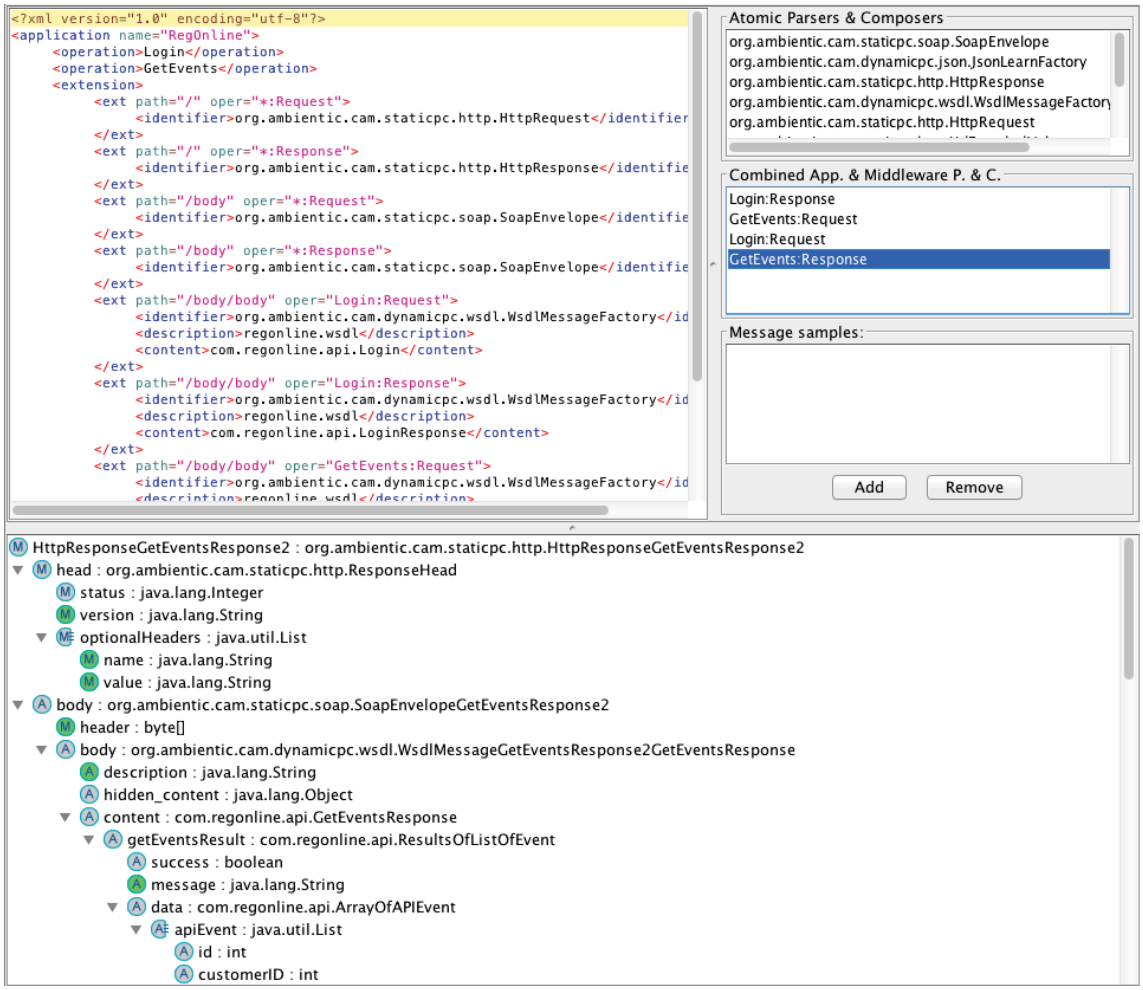


Figure 5.8 – CCL message translator generator interface.

```
<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>
```

Listing 5.1 – Fragment of the Message Model for the Regonline component. The full description can be found in the Appendix C.1.

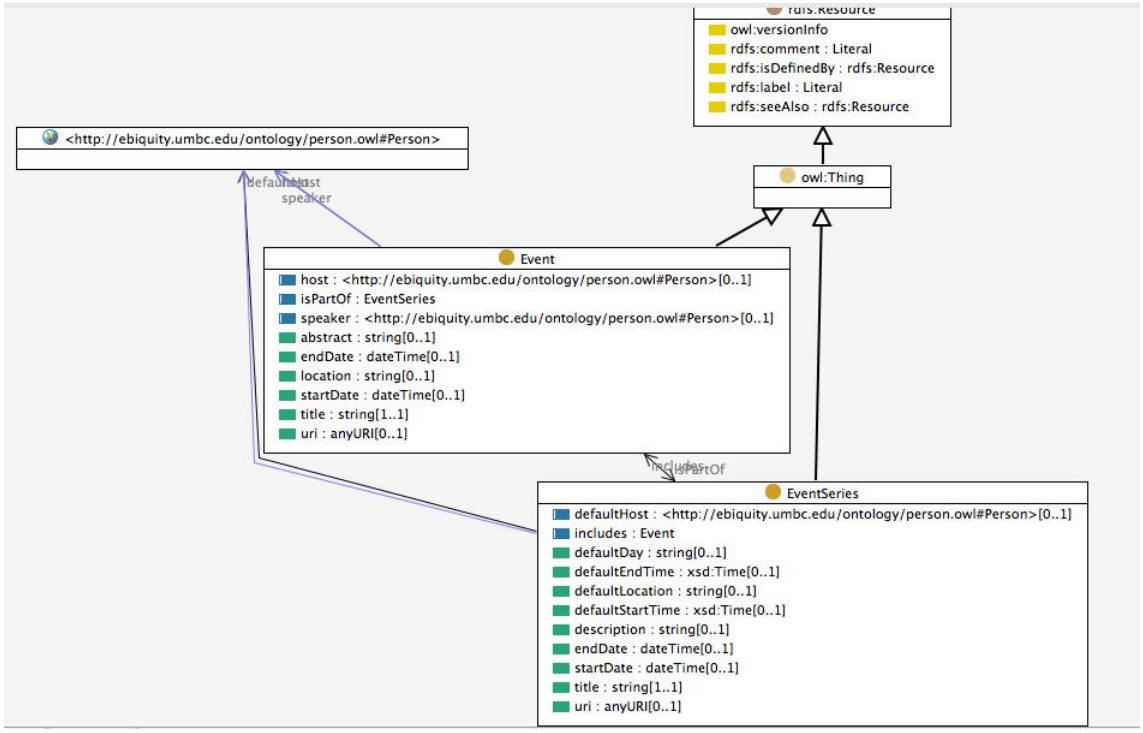


Figure 5.9 – Fragment of the OWL ontology used in the Amiando - Regonline scenario

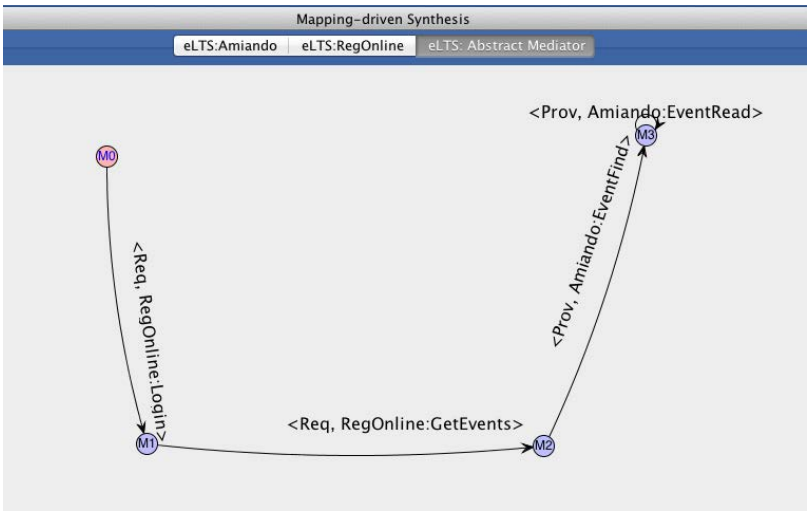


Figure 5.10 – Regonline/Amiando mediator generated by the MICS tool

Chapter 6

Conclusions

Contents

After thirty years[1] of research, thousands of proposed communication standards and protocol implementations, *functionally-compatible* networked systems still lack the fundamental feature of seamless interoperability. At the same time, research on the topic of dynamic synthesis of protocol mediators has shown that it is possible for applications to adapt to unknown protocols. However, current state of the art approaches can only synthesise mediators for very basic, and sometimes artificially simplified use-cases. In order to allow the mediation of more complex protocols and protocol stacks, there is the need to break down the problem of mediator synthesis, into clearly defined phases and explore them in conjunction. This was the goal of the FP7[2] CONNECT (`http://www.connect-forever.eu`) project, started in 2009 and which lasted for 46 months. The author joined CONNECT at the end of 2010 and also continued research beyond the end of the project. We recall that, as defined in CONNECT, the process of dynamic protocol mediation involves: (i) Discovery, (ii) Compatibility checking, (iii) Building of abstract system models, (iv) Mediator synthesis, (v) Data adaptation, (vi) Deployment and (vii) Monitoring.

In this thesis we focused on the phase of Data adaptation. However, the phases mentioned above are tightly coupled making it impossible to consider this phase outside of a

[1] The Internet Engineering Task Force (IETF) was formed on January 16, 1986 with the stated goal of "creating voluntary standards to maintain and improve the usability and interoperability of the Internet".

[2] Framework Programmes for Research and Technological Development. The FP7 is a funding program created by the European Union, for the period between 2007 to 2013.

complete mediation framework. To that end, we presented in this thesis a protocol mediation framework that allows the automation of data adaptation. To give a clear picture of the contributions presented in this thesis, we further give a chronological summary of our work.

6.1 Summary of contributions

Interoperable multimedia streaming on mobile platforms: In order to fully understand the theoretical foundations of protocol mediation, in 2010, we begun investigating a domain-specific class of protocols used for multimedia streaming. To put this research into a realistic use-case we studied the problem of interoperable multimedia streaming protocols on current generation on mobile platforms (Android and iOS). For this reason, our initial contribution, presented in in Chapter 3, focuses on multimedia streaming protocol mediation and also takes into account a number of mobility-related aspects, such as communication heterogeneity and stream scalability on mobile networks. The contribution of this work is two-fold. On the one-side, we experimentally confirmed that it is feasible to perform real-time protocol mediation on the mobile nodes, while relying on the multimedia stacks provided by each platform. On the other side, we showed the data adaptation process which is highly specific to multimedia container formats can be integrated with the overall CONNECT philosophy of synthesising and executing dynamic protocol mediators.

Cross-layer protocol mediation: Following this research, we identified a separate and much broader interoperability challenge when dealing with multilayer protocols (i.e. protocol stacks), which we called *cross-layer protocol interoperability*. Motivated by the fact that all encountered solutions fail to address this problem, being considered as a merely technical aspect, we decided to better investigate it. As we detailed in Chapter 5, synthesising mediators which can cope with the existence of multiple data-encapsulations and multiple protocol layers is essential for enabling complex systems interoperability. Also in Chapter 5, we introduced a framework architecture which, combined with state of the art methods of interface mapping, can achieve interoperability of functionally-compatible systems in the presence of cross-layer data dependencies. However, the vision of cross-layer protocol mediation is only possible if we find an efficient way to actually parse messages that consist of multiple data-encapsulations. For this reason, we only published this work in 2015 [17] about two years after we sketched the architecture of a protocol interoperability framework, taking into account cross-layer data dependencies.

Reusing and composing message translators (i.e., parsers and composers): Many interoperability solutions rely on developers to specify or implement message translators for each protocol they intend to support. Such components transform heterogeneous message formats, used by the protocols, into uniform data formats (e.g., XML, JSON) that can be efficiently managed by an interoperability framework. However, this development cost is unwanted in a field of research where we defend the need for dynamic (i.e., run-time) synthesis of protocol mediators. With this conviction, we started looking for solutions of either (i) automatically reverse-engineering message translators or (ii) efficiently reusing pre-compiled message translators. We were surprised to find that research in the domain of *Security*, the branch of automated *Packet Inspection*, proposed more elaborated solutions on the problem of message parsing than solutions in the domain of Protocol Interoperability. We borrowed ideas from both fields, and devised a solution reusing pre-compiled message translators, that we presented in Chapter 4.

Inferring the abstract data types of composite message translators: As soon as we found a solution for translating multi-layer messages, with low development effort, we also realised that the abstract data formats on the "other side" of the translation also need to be composed, knowing that in all protocols, data included in messages is arranged hierarchically. For instance, most middleware message formats (e.g., HTTP, SOAP) define a "body" message part, which has the role of containing payload data of either an application or of another middleware protocol. Although protocols are designed to combine in stacks, message data formats are not restricted to this concept. For this reason, we say data included in messages is arranged hierarchically. Thus, whenever translators are combined, we found it logical that abstract data formats should also be combined. While this problem seemed trivial at the first glance, we soon realised it is equivalent to inferring the output schema (or the data type) of a tree transformation. As far as we know, this problem has not yet been solved, while it is known that, in general, a transformation might not be recognisable by a schema [7]. In the second part of Chapter 4 we provide a formal mechanism, using tree automata, that generates an associated AST *data-schema* for an arbitrary translator composition. This contribution enables the inference of correct data-schemas, relieving developers from the time-consuming task of defining them. On a more general note, the provided method solves the type inference problem for the *substitution* class of tree compositions in linear time on the size of the output. The provided inference algorithm can thus be adapted to a number of applications beyond the scope of this work, such as XML Schema inference for XSLT transformations. We consider this single algorithm to be the most valuable contribution of this thesis, as it spans beyond the field of Protocol Interoperability, with applications in many other areas or research.

6.2 Perspectives for future work

As an ending for this thesis, we briefly introduce four possible directions of extending and using results presented in this documents.

Universal data adaptation pipeline for multimedia streaming: Today, there exist a limited number of software that allow multimedia stream adaptation. Distributed as open-source, projects such as FFmpeg, Libav, GStreamer are powerful and very complex tools. They support most of the multimedia formats used by the industry and can assure their interoperability. We believe that the DSL-based descriptions inspired by our approach as well as the overall AmbiStream architecture could significantly lower the complexity of such multimedia conversion pipelines allowing platforms that process multimedia content to better interoperate.

Cross-layer protocol mediation and beyond: In this thesis, we coined the term of "cross-layer protocol interoperability". The main goal was to differentiate from approaches that try to solve protocol interoperability layer by layer from the ones that try to solve the problem from the position of an "external observer", without in-depth knowledge of how protocols are internally governed by abstract layers and components. We studied primarily the implications of "cross-layer" dependencies on message formats. There are many other aspects that must be taken into account when analysing protocols in a "black box" manner. These directions of research include protocol behaviour, compatibility checking, data fusion, data mapping, etc.

On reusing message translators: In our solution of reusing pre-compiled message translators, we made the assumption that each translator has an associated (and accessible) abstract data schema (or data model). While we argued that such a model can be automatically obtained from precompiled components using methods such as bytecode inspection and reflection, we strongly believe that this subject needs further exploration. This is because the quality and accuracy of the obtained data model is proportional with the quality of the data mapping between interoperating systems, and thus proportional with the quality of the synthesised protocol mediator.

Type inference for multiple classes of tree transformations: The *substitution class of tree compositions* is powerful enough to model the composition of data models for communication protocols. We believe that our initial results are promising enough to motivate the exploration of other classes of tree grammar compositions, and, why not, arbitrary tree

transformations. Further results on this problem have the potential to innovate many fields and applications including XML databases and dynamically-typed programming languages.

Bibliography

[1] National, *The 9/11 Commission Report: Final Report of the National Commission on Terrorist Attacks Upon the United States (Indexed Hardcover, Authorized Edition).* W. W. Norton & Company, Aug. 2004.

[2] M. Gallaher, A. O'Connor, J. Dettbarn, and L. Gilday, *Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry.* U.S. Department of Commerce, Technology Administration, NIST, 2004. [Online]. Available: http://books.google.fr/books?id=BvSfGwAACAAJ

[3] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[4] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadis, M. Autili, M. A. Gerosa, and A. B. Hamida, "Service-oriented middleware for the future internet: state of the art and research directions," *J. Internet Services and Applications*, vol. 2, no. 1, pp. 23–45, 2011. [Online]. Available: http://dx.doi.org/10.1007/s13174-011-0021-3

[5] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," in *ICSE*, 1995.

[6] V. Issarny, A. Bennaceur, and Y. D. Bromberg, "Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability," in *LNCS SFM-11*, 2011.

[7] T. Milo and D. Suciu, "Type inference for queries on semistructured data," in *PODS*, 1999. [Online]. Available: http://doi.acm.org/10.1145/303976.303998

[8] Y. Papakonstantinou and V. Vianu, "DTD Inference for Views of XML Data," in *PODS*, 2000. [Online]. Available: http://doi.acm.org/10.1145/335168.335173

[9] V. Benzaken, G. Castagna, and A. Frisch, "Cduce: An xml-centric general-purpose language," in *Proc. of ACM SIGPLAN ICFP '03*, 2003.

[10] G. Castagna, H. Im, K. Nguyen, and V. Benzaken, "A core calculus for XQuery 3.0," 2013, http://www.pps.univ-paris-diderot.fr/~gc/papers/xqueryduce.pdf.

[11] V. Issarny, B. Steffen, B. Jonsson, G. Blair, P. Grace, M. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta, "CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems," in *14th IEEE International Conference on Engineering of Complex Computer Systems*, Postdam, Germany, Jun. 2009, pp. 154–161. [Online]. Available: https://hal.inria.fr/inria-00392809

[12] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems," in *Middleware*, 2011.

[13] A. Bennaceur, G. Blair, F. Chauvel, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, A. Pathak, R. Spalazzese, B. Steffen, B. Souville, and H. Gang, "Towards an architecture for runtime interoperability," in *ISoLA 2010 - 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Crete, Greece, 2010, pp. 206–220. [Online]. Available: https://hal.inria.fr/inria-00512446

[14] E.-M. Andriescu, A. Bennaceur, G. S. Blair, A. Calabro, P. Grace, G. Huang, V. Issarny, M. Itria, Y. Ma, C. Morisset, V. Nundloll, P.-G. Raverdy, R. Saadi, R. Speicys Cardoso, and D. Sykes, "Final CONNECT Architecture," Rapport de recherche, Dec. 2012. [Online]. Available: http://hal.inria.fr/hal-00796387

[15] A. Bennaceur, "Dynamic Synthesis of Mediators in Ubiquitous Environments," Ph.D. dissertation, Pierre and Marie Curie University (UPMC), Paris, 2013.

[16] E. M. Andriescu, T. Martinez, and V. Issarny, "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, Apr. 2015. [Online]. Available: https://hal.inria.fr/hal-01097389

[17] A. Bennaceur, E. Andriescu, R. Speicys Cardoso, and V. Issarny, "A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet," *Journal of Internet Services and Applications*, p. 14, 2015. [Online]. Available: https://hal.inria.fr/hal-01152426

[18] E. M. Andriescu, R. Speicys Cardoso, and V. Issarny, "AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices," in *ACM/IFIP/USENIX 12th International Middleware Conference*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec and F. Kon, Eds., vol. 7049. Lisbon, Portugal: Springer, Dec. 2011. [Online]. Available: http://hal.inria.fr/hal-00639633

[19] E. Andriescu, A. Bennaceur, G. S. Blair, A. Calabro, R. Speicys Cardoso, L. Cavallaro, N. Georgantas, P. Grace, V. Issarny, Y. Ma, M. Merten, N. Nostro, V. Nundloll, P. Guillaume Raverdy, R. Saadi, and D. Sykes, "Revised CONNECT Architecture," Rapport de recherche, Feb. 2012. [Online]. Available: http://hal.inria.fr/hal-00695581

[20] S. Vinoski, "Advanced Message Queuing Protocol," *Internet Computing, IEEE*, vol. 10, no. 6, pp. 87–89, 2006.

[21] N. Deakin, "JSR-343 JavaTM Message Service 2.0," May 2013, http://www.jcp.org/en/jsr/detail?id=343.

[22] S. S. Lam, "Protocol conversion," *IEEE Trans. Softw. Eng.*, vol. 14, no. 3, pp. 353–362, Mar. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.4655

[23] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller, "Automatic generation of network protocol gateways," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09, Dec. 2009.

[24] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM TOPLAS*, vol. 19, no. 2, 1997.

[25] L. Burgy, L. Reveillere, J. Lawall, and G. Muller, "Zebu: A language-based approach for network protocol message processing," *IEEE TSE*, vol. 37, no. 4, pp. 575 –591, 2011.

[26] Y.-D. Bromberg, P. Grace, and L. Réveillère, "Starlink: runtime interoperability between heterogeneous middleware protocols." in *Proceedings of 31th International Conference on Distributed Computing Systems, ICDCS (IEEE).*, Jun. 2011.

[27] N. Borisov, D. J. Brumley, and H. J. Wang, "A Generic Application-Level Protocol Analyzer and its Language," in *NDSS*, 2007.

[28] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran, "A bridging framework for universal interoperability in pervasive systems," *International Conference on Distributed Computing Systems*, Jul. 2006.

[29] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[30] Y. D.Bromberg, P. Grace, and L. Réveillère, "Starlink: Runtime Interoperability between Heterogeneous Middleware Protocols," in *ICDCS*, 2011.

[31] D. Chappell, *Enterprise service bus.* O'reilly Media, 2004.

[32] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, 1992.

[33] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.

[34] B. Spitznagel and . Garlan, "A compositional formalization of connector wrappers," in *ICSE*, 2003.

[35] R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE Transactions Software Engineering*, vol. 38, no. 4, pp. 755–777, 2012.

[36] P. Inverardi and M. Tivoli, "Automatic synthesis of modular connectors via composition of protocol mediation patterns," in *Proc. of the 35th International Conference on Software Engineering, ICSE*, 2013, pp. 3–12.

[37] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "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*, 2014, pp. 688–699.

[38] A. Bennaceur and V. Issarny, "Automated synthesis of mediators to support component interoperability," *IEEE Transactions on Software Engineering*, 2015, to appear.

[39] Y. D. Bromberg, P. Grace, L. Réveillère, and G. S. Blair, "Bridging the Interoperability Gap: Overcoming Combined Application and Middleware Heterogeneity," in *Middleware*, 2011.

[40] E. Andriescu, A. Bennaceur, P. Inverardi, V. Issarny, R. Spalazzese, and R. Speicys-Cardoso, "Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment," Research report, CONNECT, 2012. [Online]. Available: http://hal.inria.fr/hal-00805618

[41] E. M. Andriescu, A. Bennaceur, A. Bertolino, A. Calabrò, P. Grace, M. Isberner, A. Léger, M. Merten, Y. Mhoma, P. Châtel, C. Morisset, A. Pathak, P.-G. Raverdy, R. Saadi, R. Speicys Cardoso, and D. Sykes, "Deliverable D6.4: Assessment report: Experimenting with CONNECT in Systems of Systems, and Mobile Environments," Rapport de recherche, 2013. [Online]. Available: http://hal.inria.fr/hal-00793920

[42] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti, "Machine learning for emergent middleware," in *JIMSE*, 2012.

[43] V. Issarny and A. Bennaceur, "Composing distributed systems: Overcoming the interoperability challenge," in *LNCS HATS-FMCO*. Springer Verlag, 2013.

[44] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah, "Protocol-aware matching of web service interfaces for adapter dev." in *WWW*, 2010.

[45] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *J. of Systems and Soft.*, vol. 74, no. 1, 2005.

[46] R. Mateescu, P. Poizat, and G. Salaun, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE TSE*, vol. 38, no. 4, 2012.

[47] R. Spalazzese, P. Inverardi, and V. Issarny, "Towards a Formalization of Mediating Connectors for on the Fly Interoperability," in *Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009*, Sep. 2009.

[48] L. Cavallaro, E. D. Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *ICSOC*, 2009.

[49] R. Vaculín, R. Neruda, and K. P. Sycara, "The process mediation framework for semantic web services," *International Journal of Agent-Oriented Software Engineering, IJAOSE*, vol. 3, no. 1, pp. 27–58, 2009.

[50] L. Cavallaro, E. D. Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *ICSOC*, 2009.

[51] M. Merten, B. Steffen, F. Howar, and T. Margaria, "Next Generation Learnlib," in *TACAS*, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987389.1987412

[52] V. Issarny, A. Bennaceur, and Y. Bromberg, "Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability," in *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, 2011, pp. 217–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21455-4_7

[53] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, M. Paolucci, R. Saadi, and D. Sykes, "Intermediate CONNECT Architecture," Research Report, Feb. 2011. [Online]. Available: https://hal.inria.fr/inria-00584911

[54] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "Binpac: A yacc for writing application protocol parsers," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/1177080.1177119

[55] S. Marquis, T. R. Dean, and S. Knight, "SCL: a language for security testing of network applications," in *CASCON'06*, 2005.

[56] L. Burgy, L. Reveillere, J. Lawall, and G. Muller, "Zebu: A Language-Based Approach for Network Protocol Message Processing," *IEEE TSE*, 2011.

[57] F. Risso and M. Baldi, "NetPDL: An extensible XML-based language for packet header description," *Computer Networks*, vol. 50, pp. 688–706, 2006.

[58] Int. Telecommunication Union, "Information technology — asn.1 encoding rules — specification of basic encoding rules (ber), canonical encoding rules (cer), and distinguished encoding rules (der)," ITU-T Recommendation X.690, July 2002.

[59] J. Siegel, *CORBA 3 fundamentals and programming*. John Wiley & Sons Chichester, 2000, vol. 2.

[60] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *ACM Conference on Computer and Communications Security*, 2007.

[61] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *IN 15TH SYMPOSIUM ON NETWORK AND DISTRIBUTED SYSTEM SECURITY (NDSS*, 2008.

[62] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 391–402. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455820

[63] N. Borisov, D. J. Brumley, and H. J. Wang, "A generic application-level protocol analyzer and its language," in *In NDSS*, 2007.

[64] V. Paxson *et al.*, "Flex: A fast scanner generator," *Online manual: http://www. combo. org/flex*, 1995.

[65] C. Donnelly and R. Stallman, *Bison: the Yacc-compatible parser generator*. Free Software Foundation, 1993, vol. 1.

[66] S. C. Johnson, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[67] R. Mateescu, P. Poizat, and G. Salaun, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE TSE*, vol. 38, no. 4, 2012.

[68] T. Pankowski, "Detecting semantics-preserving XML schema mappings based on annotations to OWL ontology," in *LID '11 Workshop*, 2011.

[69] Z. Bellahsene, A. Bonifati, and E. Rahm, *Schema Matching and Mapping*, ser. Series: Data-Centric Systems and Applications. Springer, 2011.

[70] A. Y. Halevy, "Technical perspective - schema mappings: rules for mixing data," *Commun. ACM*, vol. 53, no. 1, p. 100, 2010.

[71] P. Shvaiko and J. Euzenat, "A survey of schema-based matching approaches," *J. on Data Semantics*, vol. 4, 2005.

[72] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook*. Cambridge University Press, 2003.

[73] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "Sawsdl: Semantic annotations for wsdl and xml schema," *IEEE Internet Computing*, vol. 11, no. 6, pp. 60–67, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1109/MIC.2007.134

[74] G. Antoniou and F. V. Harmelen, "Web ontology language: Owl," in *Handbook on Ontologies in Information Systems*. Springer, 2003, pp. 67–92.

[75] M. Shaw, "Architectural issues in software reuse: It's not just the functionality, it's the packaging," *ACM SIGSOFT*, 1995.

[76] A. C. Schwerdfeger and E. R. Van Wyk, "Verifiable Parse Table Composition for Deterministic Parsing," in *SLE*, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12107-4_15

[77] H. Basten, "Ambiguity detection methods for context-free grammars," *Master's thesis, Universiteit van Amsterdam*, 2007.

[78] R. S. Cruz, M. S. Nunes, and J. E. Gonçalves, "A Personalized HTTP Adaptive Streaming WebTV," in *User Centric Media*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, P. Daras and O. Ibarra, Eds. Springer Berlin Heidelberg, 2010, vol. 40, pp. 227–233. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12630-7_27

[79] W. Van Lancker, D. Van Deursen, E. Mannens, and R. Van de Walle, "Implementation strategies for efficient media fragment retrieval," *Multimedia Tools and Applications*, Mar. 2011.

[80] M. Lindeberg, S. Kristiansen, T. Plagemann, and V. Goebel, "Challenges and techniques for video streaming over mobile ad hoc networks," *Multimedia Systems*, vol. 17, no. 1, Feb. 2011.

[81] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "Webrtc 1.0: Real-time communication between browsers," http://www.w3.org/TR/2013/WD-webrtc-20130910/, W3C WebRTC Working Group, September 2013, w3C Working Draft.

[82] G. Coulson, "A configurable multimedia middleware platform," *IEEE MultiMedia*, Jan. 1999.

[83] G. Coulson, G. Blair, N. Davies, P. Robin, and T. Fitzpatrick, "Supporting mobile multimedia applications through adaptive middleware," *IEEE Journal on Selected Areas in Communications*, Sep. 1999.

[84] G. Blair, "On the failure of middleware to support multimedia applications," in *Interactive Distributed Multimedia Systems and Telecommunication Services*, Oct. 2000.

[85] K. Curran and G. Parr, "A middleware architecture for streaming media over IP networks to mobile devices," in *Wireless Communications and Networking*, Mar. 2003.

[86] M. Ransburg, M. Jonke, and H. Hellwagner, "An evaluation of mobile end devices in multimedia streaming scenarios," in *Mobile Wireless Middleware, Operating Systems, and Applications*, Jul. 2010.

[87] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550 (Standard), Internet Engineering Task Force, Jul. 2003, updated by RFCs 5506, 5761, 6051, 6222. [Online]. Available: http://tools.ietf.org/html/rfc3550

[88] R. Pantos and W. May, "HTTP Live Streaming," (Internet-Draft), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://tools.ietf.org/html/draft-pantos-http-live-streaming-06

[89] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," RFC 2326 (Proposed Standard), Internet Engineering Task Force, Apr. 1998. [Online]. Available: http://tools.ietf.org/html/rfc2326

[90] J. Bocharov, Q. Burns, F. Folta, K. Hughes, A. Murching, L. Olson, P. Schnell, and J. Simmons, "The Protected Interoperable File Format (PIFF)," Microsoft Corporation, Mar. 2010.

[91] "Adobe Flash Video File Format Specification, Version 10.1," Adobe Systems Incorporated, Aug. 2010. [Online]. Available: http://download.macromedia.com/f4v/video_file_format_spec_v10_1.pdf

[92] Y.-K. Wang, R. Even, T. Kristensen, and R. Jesup, "RTP Payload Format for H.264 Video," RFC 6184 (Proposed Standard), Internet Engineering Task Force, May 2011. [Online]. Available: http://tools.ietf.org/html/rfc6184

[93] "ITU-T Rec. H.222.0 — ISO/IEC 13818-1, Generic coding of moving pictures and associated audio information," 2007. [Online]. Available: http://www.iso.org/iso/catalogue_detail?csnumber=44169

[94] E. Andriescu, A. Bennaceur, A. Bertolino, P. Grace, T. Huynh, M. Kwiatkowska, B. Jonsson, A. Léger, A. Pathak, P.-G. Raverdy, R. Saadi, R. Speicys-Cardoso, D. Sykes, and M. Tivoli, "Experiment scenarios, prototypes and report - Iteration 2," Research Report, Feb. 2012. [Online]. Available: https://hal.inria.fr/hal-00695639

[95] A. C. Schwerdfeger and E. R. Van Wyk, "Verifiable Composition of Deterministic Grammars," *PLDI*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1543135.1542499

[96] A. Moors, F. Piessens, and M. Odersky, "Parser combinators in scala," *KU Leuven, CW Reports vol:CW491.*, 2008.

[97] T. Schwentick, "Automata for XML—A Survey," *JCSS*, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jcss.2006.10.003

[98] S. DeRose and J. Clark, "XML Path Language (XPath) Version 1.0," W3C, W3C Recommendation, 1999.

[99] J. Heering, P. Klint, and J. Rekers, "Incremental Generation of Parsers," in *PLDI*, 1989. [Online]. Available: http://doi.acm.org/10.1145/73141.74834

[100] S. D. Swierstra, "Combinator parsers - from toys to tools," *ENTCS*, vol. 41, 2000.

[101] E. R. Van Wyk and A. C. Schwerdfeger, "Context-aware Scanning for Parsing Extensible Languages," in *GPCE*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1289971.1289983

[102] L. Cardelli, F. Matthes, and M. Abadi, "Extensible grammars for language specialization," in *DBPL*, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=648290.754352

[103] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree Automata Techniques and Applications," http://tata.gforge.inria.fr/, 2007, release October, 12th 2007.

[104] A. Tozawa, "Towards Static Type Checking for XSLT," in *ACM DocEng*, 2001. [Online]. Available: http://doi.acm.org/10.1145/502187.502191

[105] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM TOIT*, 2005.

[106] M. Murata, "Hedge automata: a formal model for XML schemata," 1999, http://www.xml.gr.jp/relax/hedge_nice.html.

[107] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma, "METEOR-S web service annotation framework with machine learning classification," in *SWSWPC*, 2004, pp. 137–146.

[108] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. of the International Conference on Software Engineering, ICSE*, 2008, pp. 501–510.

[109] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in *Proc. of the 32nd International Conference on Software Engineering, ICSE (2)*, 2010, pp. 179–182.

List of figures

List of tables

Appendix A

List of publications

In conference proceedings

- E.-M. Andriescu, T. Martinez, and V. Issarny, *Composing Message Translators and Inferring their Data Types using Tree Automata* in 18th International Conference on Fundamental Approaches to Software Engineering (FASE) (A. Egyed and I. Schaefer, eds.), Lecture Notes in Computer Science, (London, United Kingdom), Springer, Apr 2015.

  `https://hal.inria.fr/hal-01097389`.

- E.-M. Andriescu, R. Speicys Cardoso, and V. Issarny, *AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices* in ACM/IFIP/USENIX 12th International Middleware Conference (A.-M. Kermarrec and F. Kon, eds.), vol. 7049 of Lecture Notes in Computer Science, (Lisbon, Portugal), Springer, Dec. 2011.

  `http://hal.inria.fr/hal-00639633/PDF/AmbiStream.pdf`.

In journals

- A. Bennaceur, E.-M. Andriescu, R. Speicys Cardoso, and V. Issarny, *A Unifying Perspective on Protocol Mediation: Interoperability in the Future Internet*. Journal of Internet Services and Applications, Springer, 2015, pp.14.

  `https://hal.inria.fr/hal-01152426v2/document`

Research and technical reports

- E.-M. Andriescu, A. Bennaceur, A. Bertolino, A. Calabr'o, P. Grace, M. Isberner, A. L?eger, M. Merten, Y. Mhoma, P. Chatel, C. Morisset, A. Pathak, P.-G. Raverdy,

R. Saadi, R. Speicys Cardoso, and D. Sykes, "Experimenting with CONNECT in Systems of Systems, and Mobile Environments" Research Report, 2013.

http://hal.inria.fr/hal-00793920/PDF/CONNECT_Deliverable_D6_4.pdf.

- E.-M. Andriescu, A. Bennaceur, G. S. Blair, A. Calabro, P. Grace, G. Huang, V. Issarny, M. Itria, Y. Ma, C. Morisset, V. Nundloll, P.-G. Raverdy, R. Saadi, R. Speicys Cardoso, and D. Sykes, "Final CONNECT Architecture" research report, Dec. 2012.

  http://hal.inria.fr/hal-00796387/PDF/CONNECT_Deliverable_D1_4.pdf

- E.-M. Andriescu, A. Bennaceur, P. Inverardi, V. Issarny, R. Spalazzese, and R. Speicys-Cardoso, "Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment" research report, Dec. 2012.

  http://hal.inria.fr/hal-00805618/PDF/CONNECT_Deliverable_D3_4.pdf

- E. Grousset, V. Issarny, A. Bennaceur, A. Bertolino, D. Mulas, I. Matteucci, P. Grace, G. Blair, Y. Mhoma, P. Inverardi, R. Spalazzese, M. Tivoli, M. Merten, B. Steffen, H. Qu, M. Kwiatkowska, B. Jonson, S. Cassel, Y. Ma, P. Guillaume Raverdy, R. Speicys-Cardoso, and E. Andriescu, "Project Final Report Final Publishable Summary Report" research report, Dec. 2012.

  http://hal.inria.fr/hal-00805639/PDF/Connect_WP0_FinalReport_Summary.pdf

- E.-M. Andriescu, A. Bennaceur, G. S. Blair, A. Calabro, R. Speicys Cardoso, L. Cavallaro, N. Georgantas, P. Grace, V. Issarny, Y. Ma, M. Merten, N. Nostro, V. Nundloll, P. Guillaume Raverdy, R. Saadi, and D. Sykes, "Revised CONNECT Architecture" research report, Feb. 2012.

  http://hal.inria.fr/hal-00695581/PDF/CONNECT_deliverable_D1_3.pdf

- E.-M. Andriescu, A. Bennaceur, A. Bertolino, P. Grace, T. Huynh, M. Kwiatkowska, B. Jonsson, A. Léger, A. Pathak, P.-G. Raverdy, R. Saadi, R. Speicys-Cardoso, D. Sykes, and M. Tivoli, "Experiment scenarios, prototypes and report - Iteration 2" research report, Feb. 2012.

  http://hal.inria.fr/hal-00695639/PDF/CONNECT_deliverable_D6_3.pdf

- E.-M. Andriescu, A. Azzabi, and G. Hains, "Parallel processing of forward xpath queries: an experiment with bsml" Tech. Rep. TR-LACL-2010-11, LACL (Labo-

ratory of Algorithms, Complexity and Logic), University of Paris-Est (UPEC-Paris 12), 2010.

`http://lacl.fr/Rapports/TR/TR-LACL-2010-11.pdf`

Appendix B

AmbiStream domain specific languages

This appendix relates to the contributions presented in Chapter 3 where we present a protocol mediation approach for multimedia streaming protocols on mobile devices. The mediator is specified in the form of three DSL-based models.

- **Protocol Message Format DSL** describes the format and structure of message fields of the streaming client protocol. This model is used to synthesise message parsers for incoming messages and composers for outgoing messages.

- **Multimedia Container Format DSL** is different to the first language, as it is specifically designed to capture aspects of live multimedia streaming, by adding support for common operations such as message timing, fragmenting and multiplexing.

- **Merged Automaton DSL** (Merged Automaton), is used to specify the behaviour of the Mediator in the form of a k-coloured automaton.

B.1 XSD definition of the AmbiStream DSL for Protocol Message Formats

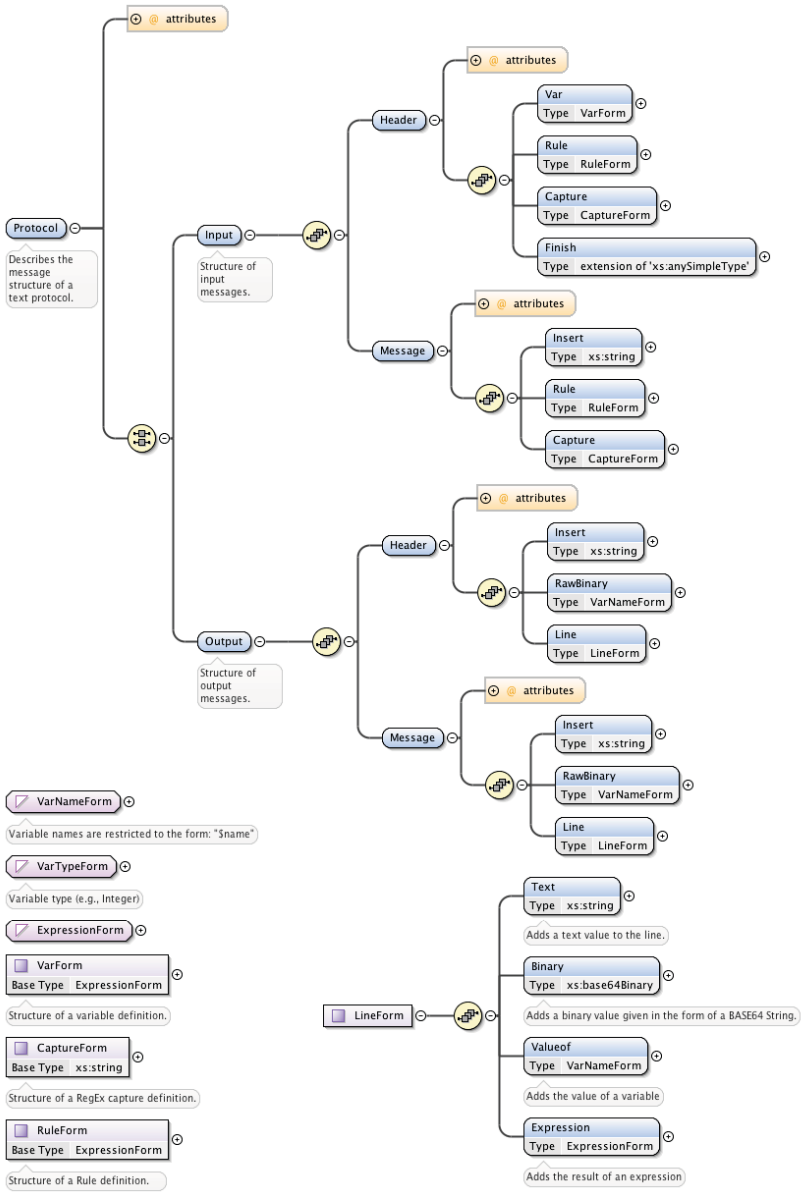


Figure B.1 – AmbiStream DSL for Protocol Message Formats. Version for text-based protocols.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.ambientic.com/AmbiStream"
  targetNamespace="http://www.ambientic.com/AmbiStream" elementFormDefault="qualified">
  <xs:simpleType name="VarNameForm">
    <xs:annotation>
      <xs:documentation>Variable names are restricted to the form: "$name"</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="$[0-9a-zA-Z]+"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="VarTypeForm">
    <xs:annotation>
      <xs:documentation>Variable type (e.g., Integer)</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="String"/>
      <xs:enumeration value="Integer"/>
      <xs:enumeration value="Bytes"/>
      <xs:enumeration value="Base64String"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ExpressionForm">
    <xs:annotation>
      <xs:documentation>e.g., $var1 + $var2</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:complexType name="VarForm">
    <xs:annotation>
      <xs:documentation>Structure of a variable definition.</xs:documentation>
    </xs:annotation>
    <xs:simpleContent>
      <xs:extension base="ExpressionForm">
        <xs:attribute name="type" default="String" use="optional" type="VarTypeForm">
          <xs:annotation>
            <xs:documentation>The type of variable (e.g., String, Integer)</xs:documentation>
          </xs:annotation>
        </xs:attribute>
        <xs:attribute name="name" use="required" type="VarNameForm">
          <xs:annotation>
            <xs:documentation/>
          </xs:annotation>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
```

```
47    </xs:complexType>
48    <xs:complexType name="CaptureForm">
49      <xs:annotation>
50        <xs:documentation>Structure of a RegEx capture definition.</xs:documentation>
51      </xs:annotation>
52      <xs:simpleContent>
53        <xs:extension base="xs:string">
54          <xs:attribute name="var" type="VarNameForm" use="required">
55            <xs:annotation>
56              <xs:documentation>Variable to store the RegEx captured value.</xs:
                    documentation>
57            </xs:annotation>
58          </xs:attribute>
59        </xs:extension>
60      </xs:simpleContent>
61    </xs:complexType>
62    <xs:complexType name="RuleForm">
63      <xs:annotation>
64        <xs:documentation>Structure of a Rule definition.</xs:documentation>
65      </xs:annotation>
66      <xs:simpleContent>
67        <xs:extension base="ExpressionForm">
68          <xs:attribute name="test" use="required">
69            <xs:simpleType>
70              <xs:restriction base="xs:string">
71                <xs:enumeration value="msg_line_delimiter"/>
72                <xs:enumeration value="msg_max_lines"/>
73                <xs:enumeration value="msg_min_lines"/>
74                <xs:enumeration value="var_capture_order"/>
75              </xs:restriction>
76            </xs:simpleType>
77          </xs:attribute>
78          <xs:anyAttribute>
79            <xs:annotation>
80              <xs:documentation>A rule can have a variable number of arguments.</xs:
                    documentation>
81            </xs:annotation>
82          </xs:anyAttribute>
83        </xs:extension>
84      </xs:simpleContent>
85    </xs:complexType>
86    <xs:complexType name="LineForm">
87      <xs:sequence>
88        <xs:element name="Text" type="xs:string">
89          <xs:annotation>
90            <xs:documentation>Adds a text value to the line.</xs:documentation>
91          </xs:annotation>
92        </xs:element>
93        <xs:element name="Binary" type="xs:base64Binary">
```

```
94        <xs:annotation>
95          <xs:documentation>Adds a binary value given in the form of a BASE64 String.</
                xs:documentation>
96        </xs:annotation>
97      </xs:element>
98      <xs:element name="Valueof" type="VarNameForm">
99        <xs:annotation>
100         <xs:documentation>Adds the value of a variable</xs:documentation>
101       </xs:annotation>
102     </xs:element>
103     <xs:element name="Expression" type="ExpressionForm">
104       <xs:annotation>
105         <xs:documentation>Adds the result of an expression</xs:documentation>
106       </xs:annotation>
107     </xs:element>
108   </xs:sequence>
109 </xs:complexType>
110 <xs:element name="Protocol">
111   <xs:annotation>
112     <xs:documentation>Describes the message structure of a text protocol.</xs:
            documentation>
113   </xs:annotation>
114   <xs:complexType>
115     <xs:all maxOccurs="1" minOccurs="1">
116       <xs:element name="Input">
117         <xs:annotation>
118           <xs:documentation>Structure of input messages.</xs:documentation>
119         </xs:annotation>
120         <xs:complexType>
121           <xs:sequence>
122             <xs:element name="Header">
123               <xs:complexType>
124                 <xs:sequence>
125                   <xs:element name="Var" type="VarForm"/>
126                   <xs:element name="Rule" type="RuleForm"/>
127                   <xs:element name="Capture" type="CaptureForm"/>
128                   <xs:element name="Finish">
129                     <xs:complexType>
130                       <xs:simpleContent>
131                         <xs:extension base="xs:anySimpleType">
132                         <xs:attribute name="test" use="required">
133                         <xs:simpleType>
134                         <xs:restriction base="xs:string">
135                         <xs:enumeration value="empty_line"/>
136                         <xs:enumeration value="text_sequence"/>
137                         <xs:enumeration value="regex_match"/>
138                         <xs:enumeration value="byte_length"/>
139                         </xs:restriction>
140                         </xs:simpleType>
```

```
                    </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Message">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Insert" type="xs:string"/>
              <xs:element name="Rule" type="RuleForm"/>
              <xs:element name="Capture" type="CaptureForm"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Output">
    <xs:annotation>
      <xs:documentation>Structure of output messages.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Header">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Insert" type="xs:string"/>
              <xs:element name="RawBinary" type="VarNameForm"/>
              <xs:element name="Line" type="LineForm"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Message">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Insert" type="xs:string"/>
              <xs:element name="RawBinary" type="VarNameForm"/>
              <xs:element name="Line" type="LineForm"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
```

```
190           </xs:complexType>
191         </xs:element>
192       </xs:all>
193       <xs:attribute fixed="text" name="type"/>
194     </xs:complexType>
195   </xs:element>
196 </xs:schema>
```

B.2 XSD definition of the AmbiStream DSL for Multimedia Container Formats

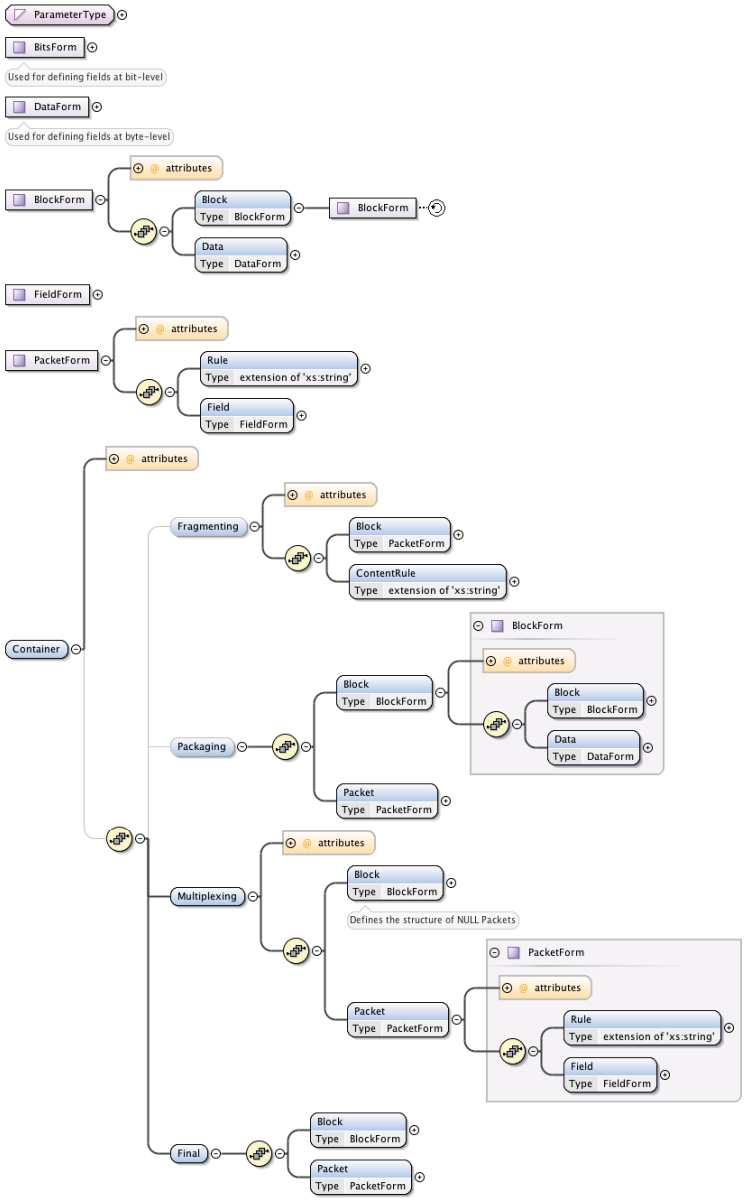


Figure B.2 – AmbiStream DSL for Multimedia Container Format

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.ambientic.com/
    AmbiStream"
  targetNamespace="http://www.ambientic.com/AmbiStream" elementFormDefault="qualified">
  <xs:simpleType name="ParameterType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="media_input_length"/>
      <xs:enumeration value="sampling_frequency"/>
      <xs:enumeration value="fragment_index"/>
      <xs:enumeration value="fragment_number"/>
      <xs:enumeration value="fragment_first"/>
      <xs:enumeration value="fragment_last"/>
      <xs:enumeration value="sequence_number"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="BitsForm">
    <xs:annotation>
      <xs:documentation>Used for defining fields at bit-level</xs:documentation>
    </xs:annotation>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="bitlength" type="xs:integer"/>
    <xs:attribute name="value" type="xs:hexBinary"/>
    <xs:attribute name="parameter" type="ParameterType"/>
  </xs:complexType>
  <xs:complexType name="DataForm">
    <xs:annotation>
      <xs:documentation>Used for defining fields at byte-level</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Bits" type="BitsForm"> </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="bytelength" type="xs:integer"/>
    <xs:attribute name="value" type="xs:hexBinary"/>
    <xs:attribute name="parameter" type="ParameterType"/>
  </xs:complexType>
  <xs:complexType name="BlockForm">
    <xs:sequence>
      <xs:element name="Block" type="BlockForm"/>
      <xs:element name="Data" type="DataForm"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="FieldForm">
    <xs:sequence>
      <xs:element name="Field" type="FieldForm"/>
      <xs:element name="InputRange">
        <xs:complexType>
          <xs:attribute name="startbyte" type="xs:integer"/>
```

```
49          <xs:attribute name="bytelength" type="xs:integer"/>
50        </xs:complexType>
51      </xs:element>
52    </xs:sequence>
53    <xs:attribute name="name" type="xs:string" use="required"/>
54    <xs:attribute name="value" type="xs:hexBinary" use="optional"/>
55  </xs:complexType>
56  <xs:complexType name="PacketForm">
57    <xs:sequence>
58      <xs:element name="Rule">
59        <xs:complexType>
60          <xs:simpleContent>
61            <xs:extension base="xs:string">
62              <xs:attribute name="type" type="xs:string"/>
63            </xs:extension>
64          </xs:simpleContent>
65        </xs:complexType>
66      </xs:element>
67      <xs:element name="Field" type="FieldForm"/>
68    </xs:sequence>
69    <xs:attribute name="name" type="xs:string"/>
70    <xs:attribute name="block" type="xs:string"/>
71  </xs:complexType>
72  <xs:element name="Container">
73    <xs:annotation>
74      <xs:documentation/>
75    </xs:annotation>
76    <xs:complexType>
77      <xs:sequence maxOccurs="1" minOccurs="0">
78        <xs:element name="Fragmenting" maxOccurs="1" minOccurs="0">
79          <xs:annotation>
80            <xs:documentation/>
81          </xs:annotation>
82          <xs:complexType>
83            <xs:sequence>
84              <xs:element name="Block" type="PacketForm"/>
85              <xs:element name="ContentRule">
86                <xs:complexType>
87                  <xs:simpleContent>
88                    <xs:extension base="xs:string">
89                      <xs:attribute name="type" type="xs:string"/>
90                    </xs:extension>
91                  </xs:simpleContent>
92                </xs:complexType>
93              </xs:element>
94            </xs:sequence>
95            <xs:attribute name="method">
96              <xs:simpleType>
97                <xs:restriction base="xs:string">
```

```
98                <xs:enumeration value="length"/>
99                <xs:enumeration value="content"/>
100               <xs:enumeration value="packaged_length"/>
101             </xs:restriction>
102           </xs:simpleType>
103         </xs:attribute>
104       </xs:complexType>
105     </xs:element>
106     <xs:element name="Packaging" maxOccurs="1" minOccurs="0">
107       <xs:annotation>
108         <xs:documentation/>
109       </xs:annotation>
110       <xs:complexType>
111         <xs:sequence>
112           <xs:element name="Block" minOccurs="1" type="BlockForm"> </xs:element>
113           <xs:element name="Packet" type="PacketForm"/>
114         </xs:sequence>
115       </xs:complexType>
116     </xs:element>
117     <xs:element name="Multiplexing">
118       <xs:annotation>
119         <xs:documentation/>
120       </xs:annotation>
121       <xs:complexType>
122         <xs:sequence>
123           <xs:element name="Block" minOccurs="1" type="BlockForm">
124             <xs:annotation>
125               <xs:documentation>Defines the structure of NULL Packets</xs:
                      documentation>
126             </xs:annotation>
127           </xs:element>
128           <xs:element name="Packet" type="PacketForm"/>
129         </xs:sequence>
130         <xs:attribute fixed="time_division" name="type" type="xs:string"/>
131         <xs:attribute name="constantbitrate" type="xs:boolean"/>
132         <xs:attribute name="timeslot" type="xs:integer"/>
133         <xs:attribute name="maxtimesloterror" type="xs:integer"/>
134       </xs:complexType>
135     </xs:element>
136     <xs:element name="Final">
137       <xs:complexType>
138         <xs:sequence>
139           <xs:element name="Block" minOccurs="1" type="BlockForm"> </xs:element>
140           <xs:element name="Packet" type="PacketForm"/>
141         </xs:sequence>
142       </xs:complexType>
143     </xs:element>
144   </xs:sequence>
145   <xs:attribute fixed="binary" name="type"/>
```

```
146       <xs:attribute name="name" type="xs:string"/>
147     </xs:complexType>
148   </xs:element>
149 </xs:schema>
```

B.3 XSD definition of the AmbiStream DSL for the Merged Automaton

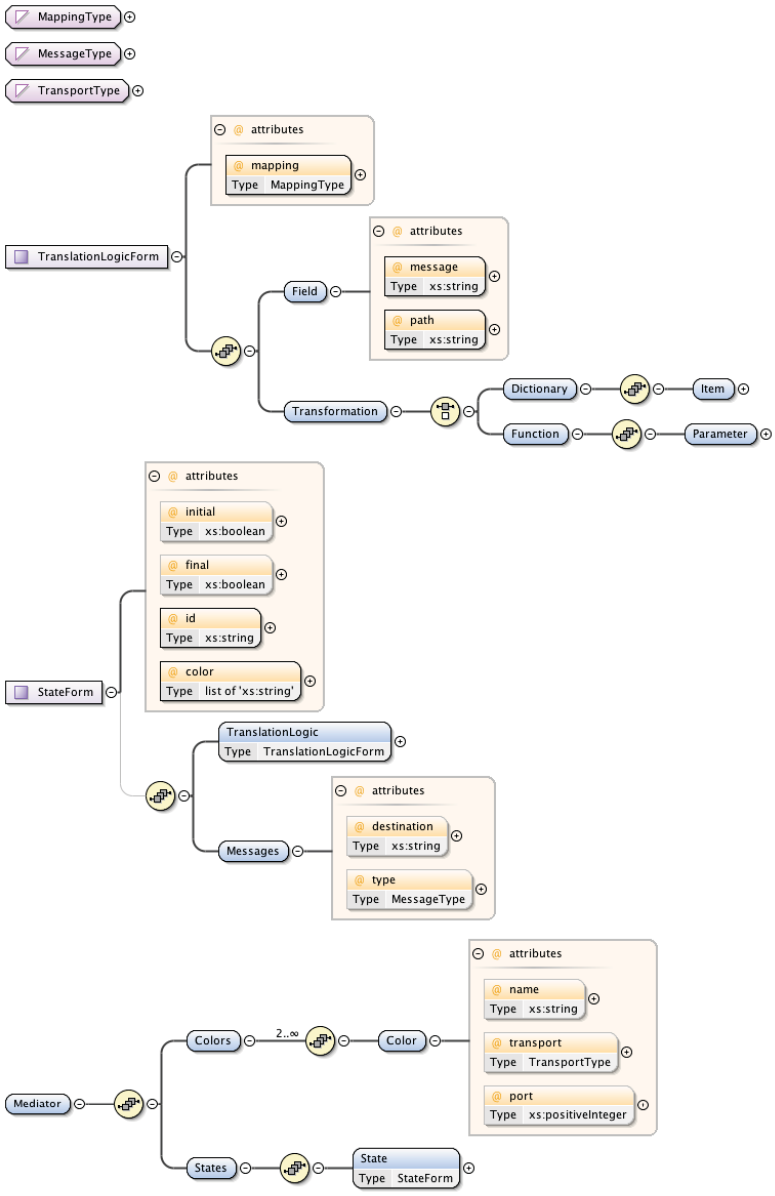


Figure B.3 – AmbiStream DSL for the Merged Automaton

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:simpleType name="MappingType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="assignment"/>
      <xs:enumeration value="dictionary"/>
      <xs:enumeration value="function"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="MessageType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="send"/>
      <xs:enumeration value="receive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="TransportType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="UDP"/>
      <xs:enumeration value="TCP"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="TranslationLogicForm">
    <xs:sequence>
      <xs:element name="Field">
        <xs:complexType>
          <xs:attribute name="message" type="xs:string" use="required"/>
          <xs:attribute name="path" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="Transformation">
        <xs:complexType>
          <xs:choice>
            <xs:element name="Dictionary">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Item">
                    <xs:complexType>
                      <xs:attribute name="key" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Function">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Parameter">
                    <xs:complexType>
```

```
                    <xs:attribute name="name" use="required"/>
                    <xs:attribute name="path" use="required"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="mapping" type="MappingType" use="required"/>
</xs:complexType>
<xs:complexType name="StateForm">
  <xs:sequence maxOccurs="1" minOccurs="0">
    <xs:element name="TranslationLogic" type="TranslationLogicForm"/>
    <xs:element name="Messages">
      <xs:complexType>
        <xs:attribute name="destination" type="xs:string"/>
        <xs:attribute name="type" type="MessageType"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="initial" type="xs:boolean"/>
  <xs:attribute name="final" type="xs:boolean"/>
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="color" use="required">
    <xs:simpleType>
      <xs:list itemType="xs:string"/>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<xs:element name="Mediator">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Colors">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded" minOccurs="2">
            <xs:element name="Color">
              <xs:complexType>
                <xs:attribute name="name" type="xs:string"/>
                <xs:attribute name="transport" type="TransportType"/>
                <xs:attribute name="port" type="xs:positiveInteger"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="States">
```

```
99          <xs:complexType>
100           <xs:sequence>
101             <xs:element name="State" type="StateForm"/>
102           </xs:sequence>
103         </xs:complexType>
104       </xs:element>
105     </xs:sequence>
106   </xs:complexType>
107  </xs:element>
108 </xs:schema>
```

B.4 Packaging phase description of the RTP Container Format using the H264 video codec.

```
<Packaging trackid="1" type="video">
  <Packet name="rtp_video">
    <Rule type="select_fragment()">any</Rule>
    <Field name="Version" value="10b">
      <InputRange startbit="0" bitlength="2"/>
    </Field>
    <Field name="Padding" value="0b">
      <InputRange startbit="2" bitlength="1"/>
    </Field>
    <Field name="Extension" value="0b">
      <InputRange startbit="3" bitlength="1"/>
    </Field>
    <Field name="CsrcCount" value="0000b">
      <InputRange startbit="4" bitlength="4"/>
    </Field>
    <Field name="Marker" value="0b">
      <InputRange startbit="8" bitlength="1"/>
    </Field>
    <Field name="Marker" value="1b">
      <!-- Set for the very last packet of the access unit indicated by the RTP
          timestamp -->
      <Rule type="select_fragment()">last</Rule>
      <InputRange startbit="8" bitlength="1"/>
    </Field>
    <Field name="PayloadType" value="96">
      <InputRange startbit="9" bitlength="7"/>
    </Field>
    <Field name="SequenceNumber" value="$sequence_number">
      <InputRange startbit="16" bitlength="16"/>
    </Field>
    <Field name="Timestamp" value="$timestamp_90KHz">
      <!-- Sampling timestamp of the content. A 90 kHz clock rate MUST be used. -->
      <InputRange startbit="32" bitlength="32"/>
    </Field>
    <Field name="Ssrc" value="$track_id">
      <!-- Synchronization Source Identifier -->
      <InputRange startbit="64" bitlength="32"/>
    </Field>
    <Field name="Payload" value="$track_id">
      <!-- RTP Payload Format for H.264 Video. See RFC6184 -->
      <InputRange startbit="96" bitlength="+"/>
      <Field name="forbidden_zero_bit" value="0b">
        <InputRange startbit="0" bitlength="1"/>
      </Field>
      <Field name="nal_ref_idc" value="10b">
```

```
45        <InputRange startbit="1" bitlength="2"/>
46      </Field>
47      <Field name="nal_ref_idc" value="11b">
48        <!-- a keyframe, a sequence parameter set or a picture parameter set -->
49        <Rule type="select_fragment()">$nal_unit_type == 5 ||
50          $nal_unit_type == 7 || $nal_unit_type == 8</Rule>
51        <InputRange startbit="1" bitlength="2"/>
52      </Field>
53      <Field name="rtp_nal_unit_type" value="$nal_unit_type">
54        <!-- Single NAL unit packet -->
55        <InputRange startbit="3" bitlength="5"/>
56      </Field>
57      <Field name="rtp_nal_unit_type" value="28">
58        <!-- Fragmented NAL unit packet -->
59        <Rule type="select_fragment()">fragmented(2)</Rule>
60        <InputRange startbit="3" bitlength="5"/>
61      </Field>
62      <Field name="FuHeader" value="28" optional="true">
63        <Rule type="select_fragment()">fragmented(2)</Rule>
64        <InputRange startbit="8" bitlength="8"/>
65        <Field name="FragmentStart" value="$fragment_first(2)">
66        </Field>
67        <Field name="FragmentEnd" value="$fragment_last(2)">
68        </Field>
69        <Field name="ReservedBit" value="0b">
70        </Field>
71      </Field>
72      <Field name="FragmentPayload" value="$fragment_data"/>
73    </Field>
74  </Packet>
75 </Packaging>
```

Appendix C

Unified mediation framework: models and generated schemas

This appendix relates to the contributions presented in Chapter 5 where we present a unified mediation framework for protocol interoperability. Below we include the following documents, part of the conference management mediation example:

- **Message Models** for the Regonlie and Amiando components. They define the strategy for assembling Atomic message translators in order to deal with the data encapsulation in different middleware solutions and cross-layer data dependencies.

- **Abstract Message Schemas**. They are an abstract description of the component's interface that facilitates the synthesis of application-layer mediators. While the scenario involved generating eight XSD schemas (one for each action of the individual protocols), we only include two which are the most relevant.

C.1 Message Models for the Regonline and Amiando components

```
<?xml version="1.0" encoding="utf-8"?>
<application name="RegOnline">
  <operation>Login</operation>
  <operation>GetEvents</operation>
  <extension>
    <ext path="/" oper="*:Request">
      <identifier>org.ambientic.cam.staticpc.http.HttpRequest</identifier>
    </ext>
    <ext path="/" oper="*:Response">
      <identifier>org.ambientic.cam.staticpc.http.HttpResponse</identifier>
    </ext>
    <ext path="/body" oper="*:*">
      <identifier>org.ambientic.cam.staticpc.soap.SoapEnvelope</identifier>
    </ext>
    <ext path="/body/body" oper="Login:Request">
      <identifier>org.ambientic.cam.dynamicpc.wsdl.WsdlMessageFactory</identifier>
      <description>regonline.wsdl</description>
      <content>com.regonline.api.Login</content>
    </ext>
    <ext path="/body/body" oper="Login:Response">
      <identifier>org.ambientic.cam.dynamicpc.wsdl.WsdlMessageFactory</identifier>
      <description>regonline.wsdl</description>
      <content>com.regonline.api.LoginResponse</content>
    </ext>
    <ext path="/body/body" oper="GetEvents:Request">
      <identifier>org.ambientic.cam.dynamicpc.wsdl.WsdlMessageFactory</identifier>
      <description>regonline.wsdl</description>
      <content>com.regonline.api.GetEvents</content>
    </ext>
    <ext path="/body/body" oper="GetEvents:Response">
      <identifier>org.ambientic.cam.dynamicpc.wsdl.WsdlMessageFactory</identifier>
      <description>regonline.wsdl</description>
      <content>com.regonline.api.GetEventsResponse</content>
    </ext>
    <ext path="/body/body/content/filter" oper="GetEvents:Request">
      <identifier>org.ambientic.cam.dynamicpc.regex.RegexMessageFactory</identifier>
      <description><![CDATA[ (?<function>[\w\.]+)\(\"(?<title>[\w]+)\"\)]]></description
          >
    </ext>
  </extension>
  <rules>
    <valuerestrict path="/head/uri" oper="*:Request">
      <enumeration value="/api/default.asmx"/>
    </valuerestrict>
    <valuerestrict path="/body/body/content/filter/function" oper="GetEvents:Request">
```

```
      <enumeration value="Title.Contains"/>
    </valuerestrict>
    <valuerestrict path="/body/body/content/username" oper="Login:Request">
      <enumeration value="my_username"/>
    </valuerestrict>
    <valuerestrict path="/body/body/content/password" oper="Login:Request">
      <enumeration value="my_password"/>
    </valuerestrict>
    <extract path="/head/headers[name=APIToken]/value" oper="GetEvents:Request">
      <fielddef>apiToken</fielddef>
    </extract>
    <extract path="/head/headers[name=SOAPAction]/value" oper="*:Request">
      <fielddef>soapAction</fielddef>
    </extract>
    <noderestrict path="/head/apiToken" oper="GetEvents:Request">
      <replayonly>true</replayonly>
      <appscope>true</appscope>
    </noderestrict>
    <noderestrict path="/body/body/content" oper="GetEvents:Request">
      <optional>true</optional>
    </noderestrict>
    <map path="/head/soapAction" oper="*:Request">
      <source>/body/body/soapAction</source>
    </map>
  </rules>
  <concepts>
    <annotation path="/head/apiToken" oper="GetEvents:Request">
      <concept>SecurityToken</concept>
    </annotation>
    <annotation path="/body/body/content/loginResult/data/eventSessionId" oper="Login:
        Response">
      <concept>SecurityToken</concept>
    </annotation>
    <annotation path="/body/body/content/data" oper="GetEvents:Response">
      <concept>Event</concept>
    </annotation>
    <annotation path="/body/body/content/filter/title" oper="GetEvents:Request">
      <concept>EventTitle</concept>
    </annotation>
  </concepts>
</application>
```

```
<?xml version="1.0" encoding="utf-8"?>
<application name="Amiando">
  <operation>EventFind</operation>
  <operation>EventRead</operation>
  <extension>
    <ext path="/" oper="*:Request">
      <identifier>org.ambientic.cam.staticpc.http.HttpRequest</identifier>
```

```
8     </ext>
9     <ext path="/" oper="*:Response">
10      <identifier>org.ambientic.cam.staticpc.http.HttpResponse</identifier>
11    </ext>
12    <ext path="/body" oper="EventFind:Response">
13      <identifier>org.ambientic.cam.dynamicpc.json.JsonLearnFactory</identifier>
14      <description>amiandoeventfind_01.json</description>
15      <package>com.amiando.eventfind</package>
16    </ext>
17    <ext path="/body" oper="EventRead:Response">
18      <identifier>org.ambientic.cam.dynamicpc.json.JsonLearnFactory</identifier>
19      <description>amiandoeventread_01.json</description>
20      <package>com.amiando.eventread</package>
21    </ext>
22    <ext path="/head/uri" oper="*:Request">
23      <identifier>org.ambientic.cam.staticpc.http.UrlEncodedValues</identifier>
24    </ext>
25  </extension>
26  <rules>
27    <valuerestrict path="/head/uri/uri" oper="EventFind:Request">
28      <enumeration value="/api/event/find"/>
29    </valuerestrict>
30    <valuerestrict path="/head/uri/version" oper="*:Request">
31      <enumeration value="1"/>
32    </valuerestrict>
33    <valuerestrict path="/head/uri/format" oper="*:Request">
34      <enumeration value="json"/>
35    </valuerestrict>
36    <valuerestrict path="/head/uri/uri" oper="EventRead:Request">
37      <pattern value="/api/event/([0-9])+"/>
38    </valuerestrict>
39    <valuerestrict path="/head/uri/apikey" oper="*:Request">
40      <enumeration value="Fdiri3ncNyLtB**********************GEkfkog6Tw21Wo"/>
41    </valuerestrict>
42    <extract path="/head/headers[name=apikey]/value" oper="*:Request">
43      <fielddef>apikey</fielddef>
44    </extract>
45    <extract path="/head/headers[name=version]/value" oper="*:Request">
46      <fielddef>version</fielddef>
47    </extract>
48    <extract path="/head/headers[name=format]/value" oper="*:Request">
49      <fielddef>format</fielddef>
50    </extract>
51    <extract path="/head/headers[name=title]/value" oper="EventFind:Request">
52      <fielddef>title</fielddef>
53    </extract>
54    <noderestrict path="/head/uri/uri" oper="EventRead:Request">
55      <appscope>true</appscope>
56    </noderestrict>
```

```
57      <noderestrict path="/head/uri/apikey" oper="*:Request">
58        <optional>true</optional>
59      </noderestrict>
60    </rules>
61    <concepts>
62      <annotation path="/head/uri/uri" oper="EventRead:Request">
63        <concept>EventID</concept>
64      </annotation>
65      <annotation path="/body/content/ids" oper="EventFind:Response">
66        <concept>EventID</concept>
67      </annotation>
68      <annotation path="/head/uri/title" oper="EventFind:Request">
69        <concept>EventTitle</concept>
70      </annotation>
71      <annotation path="/body/content/event" oper="EventRead:Response">
72        <concept>Event</concept>
73      </annotation>
74    </concepts>
75  </application>
```

C.2 Abstract Message Schemas

C.2.1 Regonline GetEventsResponse

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
        targetNamespace="fr.inria.arles.mediation.com.regonline" xmlns:xsi="http://www.w3.
        org/2001/XMLSchema-instance" xmlns:ns1="fr.inria.arles.mediation.com.regonline">
3     <xs:import namespace="http://www.w3.org/2001/XMLSchema-instance"/>
4
5     <xs:element name="GetEventsResponse">
6       <xs:complexType>
7         <xs:sequence minOccurs="0">
8           <xs:element name="GetEventsResponse" type="ns1:GetEventsResponse"/>
9         </xs:sequence>
10      </xs:complexType>
11    </xs:element>
12
13    <xs:complexType name="GetEventsResponse">
14      <xs:sequence minOccurs="0">
15        <xs:element name="GetEventsResult" type="ns1:GetEventsResult"/>
16      </xs:sequence>
17    </xs:complexType>
18
19    <xs:complexType name="GetEventsResult">
20      <xs:sequence>
21        <xs:element ref="ns1:Success"/>
```

```
22      <xs:element ref="ns1:Data"/>
23    </xs:sequence>
24  </xs:complexType>
25
26  <xs:element name="Success" type="xs:boolean"/>
27  <xs:element name="Data">
28    <xs:complexType>
29      <xs:sequence>
30        <xs:element ref="ns1:APIEvent"/>
31      </xs:sequence>
32    </xs:complexType>
33  </xs:element>
34  <xs:element name="APIEvent" type="ns1:APIEvent"/>
35  <xs:complexType name="APIEvent">
36    <xs:complexContent mixed="false">
37      <xs:extension base="ns1:EventCommonFields">
38        <xs:sequence>
39          <xs:element minOccurs="1" maxOccurs="1" name="ID" type="xs:int" />
40          <xs:element minOccurs="1" maxOccurs="1" name="CustomerID" type="xs:int" />
41          <xs:element minOccurs="0" maxOccurs="1" name="ParentID" nillable="true" type="
                xs:int" />
42          <xs:element minOccurs="0" maxOccurs="1" name="Status" type="xs:string" />
43          <xs:element minOccurs="0" maxOccurs="1" name="Title" type="xs:string" />
44          <xs:element minOccurs="1" maxOccurs="1" name="StartDate" nillable="true" type=
                "xs:dateTime" />
45          <xs:element minOccurs="1" maxOccurs="1" name="EndDate" nillable="true" type="
                xs:dateTime" />
46          <xs:element minOccurs="1" maxOccurs="1" name="ActiveDate" nillable="true" type
                ="xs:dateTime" />
47          <xs:element minOccurs="0" maxOccurs="1" name="ClientEventID" type="xs:string"
                />
48          <xs:element minOccurs="1" maxOccurs="1" name="TypeID" nillable="true" type="xs
                :int" />
49          <xs:element minOccurs="0" maxOccurs="1" name="Type" type="xs:string" />
50          <xs:element minOccurs="0" maxOccurs="1" name="City" type="xs:string" />
51          <xs:element minOccurs="0" maxOccurs="1" name="State" type="xs:string" />
52          <xs:element minOccurs="0" maxOccurs="1" name="Country" type="xs:string" />
53          <xs:element minOccurs="0" maxOccurs="1" name="PostalCode" type="xs:string" />
54          <xs:element minOccurs="0" maxOccurs="1" name="LocationName" type="xs:string"
                />
55          <xs:element minOccurs="0" maxOccurs="1" name="LocationRoom" type="xs:string"
                />
56          <xs:element minOccurs="0" maxOccurs="1" name="LocationPhone" type="xs:string"
                />
57          <xs:element minOccurs="0" maxOccurs="1" name="LocationBuilding" type="xs:
                string" />
58          <xs:element minOccurs="0" maxOccurs="1" name="LocationAddress1" type="xs:
                string" />
```

```
59          <xs:element minOccurs="0" maxOccurs="1" name="LocationAddress2" type="xs:
                string" />
60          <xs:element minOccurs="0" maxOccurs="1" name="TimeZone" type="xs:string" />
61          <xs:element minOccurs="0" maxOccurs="1" name="Capacity" nillable="true" type="
                xs:int" />
62          <xs:element minOccurs="0" maxOccurs="1" name="CurrencyCode" type="xs:string"
                />
63          <xs:element minOccurs="0" maxOccurs="1" name="Keywords" type="xs:string" />
64          <xs:element minOccurs="1" maxOccurs="1" name="AddDate" type="xs:dateTime" />
65          <xs:element minOccurs="0" maxOccurs="1" name="AddBy" type="xs:string" />
66          <xs:element minOccurs="1" maxOccurs="1" name="ModDate" type="xs:dateTime" />
67          <xs:element minOccurs="0" maxOccurs="1" name="ModBy" type="xs:string" />
68          <xs:element minOccurs="0" maxOccurs="1" name="Channel" type="xs:string" />
69          <xs:element minOccurs="1" maxOccurs="1" name="IsWaitlisted" type="xs:boolean"
                />
70          <xs:element minOccurs="0" maxOccurs="1" name="Culture" type="xs:string" />
71          <xs:element minOccurs="0" maxOccurs="1" name="MediaType" type="xs:string" />
72          <xs:element minOccurs="1" maxOccurs="1" name="IsActive" type="xs:boolean" />
73          <xs:element minOccurs="1" maxOccurs="1" name="IsOnSite" type="xs:boolean" />
74          <xs:element minOccurs="1" maxOccurs="1" name="Latitude" nillable="true" type="
                xs:decimal" />
75          <xs:element minOccurs="1" maxOccurs="1" name="Longitude" nillable="true" type=
                "xs:decimal" />
76          <xs:element minOccurs="0" maxOccurs="1" name="FloorMap" type="xs:string" />
77          <xs:element minOccurs="1" maxOccurs="1" name="TotalRevenue" nillable="true"
                type="xs:decimal" />
78          <xs:element minOccurs="1" maxOccurs="1" name="TotalRegistrations" nillable="
                true" type="xs:int" />
79          <xs:element minOccurs="1" maxOccurs="1" name="TotalCancels" nillable="true"
                type="xs:int" />
80          <xs:element minOccurs="1" maxOccurs="1" name="TotalSubstitutions" nillable="
                true" type="xs:int" />
81          <xs:element minOccurs="1" maxOccurs="1" name="TargetAttendance" nillable="true
                " type="xs:int" />
82          <xs:element minOccurs="1" maxOccurs="1" name="TotalIncompletes" nillable="true
                " type="xs:int" />
83        </xs:sequence>
84      </xs:extension>
85    </xs:complexContent>
86  </xs:complexType>
87
88  <xs:complexType name="EventCommonFields" />
89 </xs:schema>
```

C.2.2 Amiando ReadResponse

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
```

```
  targetNamespace="fr.inria.arles.mediation.com.amiando"
  xmlns:ns1="fr.inria.arles.mediation.com.amiando">
  <xs:element name="EventReadResponse" type="ns1:EventReadResponse"/>

  <xs:complexType name="EventReadResponse">
    <xs:sequence>
      <xs:element name="event" type="ns1:event"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="event">
    <xs:sequence>
      <xs:element name="lastModified" type="xs:dateTime"/>
      <xs:element name="selectedDate" type="xs:dateTime"/>
      <xs:element name="visibility" type="xs:NCName"/>
      <xs:element name="location" type="xs:string"/>
      <xs:element name="street" type="xs:NCName"/>
      <xs:element name="publishSearchEngines" type="xs:boolean"/>
      <xs:element name="hostId" type="xs:integer"/>
      <xs:element name="eventType" type="xs:NCName"/>
      <xs:element name="country" type="xs:NCName"/>
      <xs:element name="city" type="xs:NCName"/>
      <xs:element name="id" type="xs:integer"/>
      <xs:element name="title" type="xs:NCName"/>
      <xs:element name="timezone" type="xs:string"/>
      <xs:element name="organisatorDisplayName" type="xs:string"/>
      <xs:element name="creationTime" type="xs:dateTime"/>
      <xs:element name="longitude" type="xs:decimal"/>
      <xs:element name="latitude" type="xs:decimal"/>
      <xs:element name="language" type="xs:NCName"/>
      <xs:element name="identifier" type="xs:NCName"/>
      <xs:element name="selectedEndDate" type="xs:dateTime"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```