Dynamic Synthesis of Mediators in Ubiquitous Environments

Amel Bennaceur

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Amel BENNACEUR

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Synthèse dynamique de médiateurs dans les environnements ubiquitaires

Dynamic Synthesis of Mediators in Ubiquitous Environments

soutenance prévue le 18 juillet 2013

devant le jury composé de :

Pr. Benoît Baudry	Rapporteur
Pr. Elisabetta Di Nitto	Rapporteur
Pr. Paola Inverardi	Examinateur
Pr. Sébastien Tixeuil	Examinateur
Dr. Yérom-David Bromberg	Examinateur
Pr. Valérie Issarny	Directrice de thèse

Abstract

Given today's highly dynamic and extremely heterogeneous software systems, automatically achieving interoperability between their software components —without modifying them— is more than simply desirable, it is fast becoming a necessity. Although much work has been carried out on interoperability, existing solutions have not fully succeeded in keeping pace with the increasing complexity and heterogeneity of modern software, and meeting the demands of runtime support. These solutions either require developers to implement *mediators*, which are software entities that reconcile the differences between the implementations of software components so as to enable them to work together, or generate mediators based on declarative specifications of the composition of components or correspondences between the components' interfaces. Due to their dependency on such specifications, existing solutions are insufficient for ubiquitous environments where software components meet dynamically and interactions take place spontaneously.

The main contribution of this thesis is to define an approach and provide a supporting tool for the automated synthesis and deployment of mediators in order to enable heterogeneous software components, with compatible functionalities, to interoperate. The synthesised mediators reconcile the differences between the interfaces of the components and coordinate their behaviours from the application down to the middleware layers.

In this thesis, we show that ontology reasoning, constraint programming, and automata techniques can provide the basis for a practical and sound solution to automate the synthesis of mediators at both design time and runtime. The full automation of mediator synthesis removes the need for solutions requiring declarative, often detailed, specifications of how to perform mediation. We validate our approach through the development of a tool, MICS, and its experimentation with a number of case studies ranging from heterogeneous chat applications to emergency management in systems of systems. Through these case studies, we demonstrate the viability and efficiency of the automated synthesis of mediators to enable functionally-compatible software components to interoperate seamlessly.

Keywords: Interoperability, Mediators, Middleware, Software Composition, Automation, Behaviour, Ontology, Reasoning.

A papa, mama et les shadis

Acknowledgements

It is not so much our friends' help that helps us as the confident knowledge that they will help us.

— Epicurus, philosopher (c. 341-270 BCE)

First and foremost, I would like to express my deep gratitude to my supervisor, Valérie, for her constant encouragement and staunch support. *Valérie, merci du fond du cœur.*

I am deeply indebted to my reviewers, Benoît Baudry and Elisabetta Di Nitto for their invaluable suggestions and feedback. I would also thank the rest of my thesis committee, Paola Inverardi, Sébastien Tixeuil, and Yérom-David Bromberg for their encouragement, insightful comments, and hard questions.

My sincere thanks go to ARLES members, past and present, for putting up with half-baked ideas and for your openness and support that has made my stay here a pleasant one. And additional thanks to Animesh, Roberto, Sara, Alessandra, Nikolaos, Nelly, Manel, Thiago, Daniel, Georgis, and Georgios for thought-provoking discussions and valuable advice.

Thank you, धन्यवाद, Obrigado, شكرا, Mercie, Grazie, Σας ευχαριστώ, ¡gracias

To the Inria RH and MIRIAD teams, especially Martine, Catherine, Miriam, Stéphanie, Florence, Sylvie, Philippe, Alain, Xavier et Isabelle: You have always been there to help make things possible. To my partners of AGOS activities: Steven, Thierry, Sylvain, Szymon, Ines, Céline, Anne-Céline, and Jelena: I really enjoyed spending the lunch breaks playing and dancing. To my friends from INI or Algiers: Akram, Souad, Insaf, Nadia, Steph, Lina: Your presence made me really feel like home. To Richard, without whom I would never had been able to write my thesis in English: you were more than a teacher, you were also a friend.

To the CONNECT team, thank you! you handled my buggy prototypes and turned them into something real. And special thanks to Gordon, Paul, Chris, Romina, Massimo(s), Charles, and Bengt.

Finally, I am inexpressibly grateful to my parents and the not-so-little brothers for their love and care and for putting up with me when my nerves were on edge.

Contents

Contents viii

List of Figures xii

List of Tables xvi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Case Studies . . . 7
- 1.3 Influences . . . 9
- 1.4 Contributions . . . 13
- 1.5 Thesis Outline . . . 15

2 Interoperability: A Landscape of the Research Field 17
- 2.1 The Software Architecture Perspective: Understanding Interoperability 19
  - 2.1.1 Formal Reasoning about Interoperability . . . 22
  - 2.1.2 Mediators to Support Interoperability . . . 27
  - 2.1.3 Mediation in Ubiquitous Computing Environments . . . 29
- 2.2 The Middleware Perspective: Implementing Mediators . . . 30
  - 2.2.1 Universal Middleware . . . 31
  - 2.2.2 Middleware Bridges . . . 32
  - 2.2.3 Service Buses . . . 34
- 2.3 The Formal Methods Perspective: Synthesising Mediators . . . 37
  - 2.3.1 Mediator Synthesis Using a Specification of the Composed System 37
  - 2.3.2 Mediator Synthesis Using a Partial Specification . . . 40
- 2.4 The Semantic Web Perspective: Mediation at Runtime . . . 43

2.4.1 Ontological Modelling and Reasoning 44
2.4.2 Semantic Web Services 46
2.4.3 Semantic Mediation Bus. 49
2.5 Summary 50

3 Achieving Eternal Interoperability: The Role of Automated Mediator Synthesis 53
3.1 The CONNECT Approach to Eternal Interoperability 54
3.2 Modelling Components 56
3.3 Emergent Middleware 59
3.4 Emergent Middleware Enablers 60
3.4.1 Discovery Enabler: Locating Components 60
3.4.2 Learning Enabler: Completing Component Models 62
3.4.3 Synthesis Enabler: Synthesising Mediators 64
3.5 Summary 66

4 Automated Synthesis of Mediators 67
4.1 The File Management Example 68
4.2 Specification of Interface Matching 72
4.2.1 One-to-One Matching 74
4.2.2 One-to-Many Matching 75
4.2.3 Many-to-Many Matching 77
4.3 Computation of Interface Matching using Constraint Programming . 79
4.3.1 Complexity of Interface Matching 80
4.3.2 Interface Matching as a Constraint Satisfaction Problem . . . 81
4.3.3 Leveraging Constraint Programming for Ontological Reasoning 82
4.4 Synthesising Correct-by-Construction Mediators 86
4.5 Summary 93

5 From Abstract to Concrete Mediators 95
5.1 The Case of the Same Middleware 96
5.1.1 From Ontological Relations to Data Translation Functions . . 98
5.1.2 Application to the File Management Example 101
5.2 The Case of Different Middleware Based on the Same Interaction Pattern 104
5.2.1 Ontology-based Modelling of Middleware Interaction Patterns 105

5.2.2 Application to the Weather Example . . . 111
5.3 The Case of Middleware Based on Different Interaction Patterns . . . 112
5.3.1 Coordination across Interaction Patterns . . . 113
5.3.2 Application to the Positioning Example . . . 116
5.4 Summary . . . 118

6 Implementation & Assessment 119
6.1 The MICS tool . . . 119
6.2 Case Studies . . . 122
6.2.1 Instant Messaging: One-to-One Matching . . . 125
6.2.2 File Management: One-to-Many Matching . . . 129
6.2.3 Purchase Order: Mediation of Semantic Web Services . . . 131
6.2.4 Event Management: Unified Application-Middleware Mediation 136
6.2.5 GMES: Runtime Mediation . . . 140
6.3 Performance of MICS . . . 142
6.4 Summary . . . 144

7 Conclusion 145
7.1 Contributions . . . 145
7.2 Future Work . . . 147
7.2.1 Mediator Synthesis as a Service . . . 147
7.2.2 Mediator Evolution . . . 150
7.3 One More Thing... . . . 151

Appendix A FSP Syntax & Semantics 153

Appendix B DL Syntax & Semantics 157

References 159

List of Figures

1.1 The different perspectives on interoperability . . . 10
1.2 Middleware . . . 11
1.3 A multifaceted approach for interoperability . . . 13

2.1 The weather example . . . 18
2.2 Interoperability barriers . . . 19
2.3 *C2*, *Weather Service*, and associated connector in *Country 1* . . . 20
2.4 *Weather Station*, its client, and associated connector in *Country 2* . . . 21
2.5 Mediator solving an architectural mismatch between *C2* and *Weather Station* . . . 27
2.6 Illustrating the use of universal middleware in the weather example . . . 31
2.7 Illustrating the use of a middleware bridge in the weather example . . . 33
2.8 Illustrating the use of an ESB in the weather example . . . 36
2.9 Illustrating the synthesis of mediators using a specification of the composed system . . . 38
2.10 Illustrating the synthesis of mediators using a partial specification of the mediator . . . 41
2.11 Extract of the OWL-S Ontology . . . 46
2.12 Extract of the WSMO Ontology . . . 48

3.1 The CONNECT approach for creating emergent middleware . . . 55
3.2 Emergent middleware between *C2* and *Weather Station* . . . 60
3.3 Emergent middleware enablers . . . 61
3.4 Overview of the discovery enabler . . . 61
3.5 Illustrating capability learning . . . 63
3.6 Learning the behaviour of *C2* . . . 64

3.7 Overview of the synthesis enabler 65

4.1 Making WebDAV client and Google Docs service interoperable 69
4.2 The file management ontology 70
4.3 One-to-one matching process: $M_{1-1}(\alpha, \overline{\beta})$ 74
4.4 One-to-many matching process: $M_{1-n}(\alpha, X_2)$ 76
4.5 Many-to-many matching process: $M_{m-n}(X_1, X_2)$ 79
4.6 Illustrating ontology encoding on an extract of the file ontology 85
4.7 Representative cases for mediator synthesis 88
4.8 Illustrating the synthesis of a mediator between *WDAV* and *GDocs* . . 92

5.1 Illustrating data translation for one-to-one matching 96
5.2 Concretising $M_{1-n}(\alpha, X_2)$ 98
5.3 Illustrating concretisation in the file management example 101
5.4 The RPC ontology with SOAP specialisation 106
5.5 The DSM ontology specialised with LIME 108
5.6 Event middleware ontology specialised with JMS 109
5.7 Illustrating concretisation in the weather example 112
5.8 Mapping interaction patterns to required/provided actions 114
5.9 Illustrating concretisation in the positioning example 117

6.1 Overview of MICS 120
6.2 Using MICS as a standalone tool 123
6.3 Making XMPP and MSNP components interoperable 125
6.4 The IM ontology 126
6.5 Latency for mediated and non-mediated interactions between IM components 128
6.6 Latency for mediated and non-mediated interactions between WebDAV and Google Docs 130
6.7 Making *Blue* and *Moon* interoperable 131
6.8 The purchase ontology 132
6.9 Making Amiando client and RegOnline service interoperable 137
6.10 The event management ontology 138
6.11 Latency for mediated and non-mediated interactions between Amiando and RegOnline 139

6.12 Illustrating interoperability in GMES 140
6.13 Latency for mediated and non-mediation interaction between GMES components 141
6.14 Comparison of the time necessary for each mediation step 143

7.1 A multifaceted approach for interoperability 146
7.2 Mediator Synthesis as a Service 148
7.3 Towards mediator evolution 150

List of Tables

2.1 Summary of approaches to interoperability 51

6.1 Summary of the case studies 124
6.2 Comparing our automated mediation solution with the solutions participating in the SWS challenge 135
6.3 Processing time (in milliseconds) for each mediation step 142

Chapter 1

Introduction

> *"BABEL! BABEL! BABEL! - Between the mind that plans and the hands that build there must be a mediator."*
>
> — in Metropolis by Fritz Lang, filmmaker (1890-1976)

How can I chat with my friend on Yahoo! using my enterprise messaging service? How can I open my Google Docs files using the Finder application on my Mac? How can a company use the same application to order products from different providers? How can the command and control centre of one country effectively use the resources offered by another country in emergency situations? All the answers come down to *interoperability*. Whether it be expressed in terms of compliance to industry standards, in terms of sharing and reusing information across different systems, or in terms of the ability of several components to work together to reach a common goal, interoperability is, and remains, a key concern in software engineering and distributed systems. The challenge we address in this thesis is to make software components interoperate seamlessly, even though they were not designed and implemented to work together, provided that the functionality required by one can be provided by the other. As software systems grow increasingly complex, heterogeneous, and dynamic, automatically achieving interoperability is fast becoming a central challenge.

In this introductory chapter, we look into the need for interoperability and show why, despite extensive interest and intensive work, interoperability remains an open problem, especially in ubiquitous computing environments. Next, we show how the various approaches for interoperability, which have been proposed across the fields of

software engineering and distributed systems, inspired us when defining a solution to enforce interoperability by means of the automated synthesis of intermediary software entities, *mediators*. We then summarise our contributions. The chapter concludes by an outline of the structure of this document.

1.1 Motivation

To software developers, life may sometimes seem like a scene from "Modern Times" where Charlie Chaplin is labouring away at an assembly line, frantically tightening bolts over and over again. Modern software systems are increasingly built by assembling, and re-assembling, existing components —possibly distributed among many devices— so as to create innovative services. Since the components of a software system are often designed and implemented independently, software developers spend a lot of time, and effort, adding pieces of code so as to allow these components to work together and satisfy the requirements of the software system. The rapid pace of technological change combined with the increasing demands for high-quality software in reduced time and at lower cost, may overwhelm developers who have to deal with a multitude of details just to make components work together. Besides being a complex and error-prone task, enabling independently-developed components to work together is both daunting and tedious. Developers should be free to spend more time creating new services and designing innovative software systems and less time tightening and re-tightening bolts. The goal of our work is to enable independently-developed software components to work together, if need be, despite the many differences in their implementations. In other words, we aim to achieve interoperability between existing software components that were not designed or implemented to operate together automatically.

Let us first explain what interoperability means. In a broad sense, interoperability simply refers to the ability of different systems (both hardware and software), organisations, and people to operate together to achieve a common goal [Tol03, KC09, JGGA+13]. In this thesis, we focus on interoperability between *software components* that may be distributed across multiple, possibly mobile, devices communicating through the network. In this context, let us consider representative definitions of interoperability from software engineering and distributed systems:

Definition 1: *"The ability of two or more systems or components to exchange information and to use the information that has been exchanged."* [IEE90, pp.42]

Definition 2: *"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."* [TVS06, pp.8]

Definition 3: *"Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context."* [BCK12, pp.104]

These definitions set out several key concepts that underlie interoperability. First, all the definitions stress that interoperability is not specific to one component but should be considered between several components. Hence, the first question is between which components should interoperability take place? Definition 2 emphasises the idea of services: interoperability takes place between components that can rely on each other's services. But the term service is a very vague one. To make it clearer, we distinguish between a macroscopic view of the service, that is the *functionality* of the component and a more refined view, that is the *interface* of the component, also referred to in Definition 3. The functionality specifies the high-level role of the component, e.g., Google Docs provides the functionality of managing files. It makes sense for two components to interoperate if the functionality required by one can be provided by the other, i.e., if the components are *functionally compatible*. The interface, on the other hand, is specific to the implementation of the component. It describes the set of actions performed by/on the component to implement its functionality and allows components to exchange information.

The second question is what does working together mean? Definitions 1 and 3 stress *useful* information exchange. First, the way a component exchanges information defines its *observable behaviour*. Specifically, the component's observable behaviour defines how the component uses the actions of its interface to achieve its functionality. Still, a software system can only satisfy its requirements if the components that form this system can successfully exchange information. Hence, the components of a software system need to combine their behaviours such that whenever one of the components requires some action, another component is ready to provide it.

Definition 2 highlights the reliance on a common standard. Let us first consider standards in the strict sense of the term, i.e., as a *"document, established by consensus and approved by a recognised body, that provides, for common and repeated use, rules, guidelines or characteristics for activities or their results, aimed at the achievement of the optimum degree of order in a given context"* [ISO11]. Internet standards, like TCP/IP, SMTP, UTF-8, XML, and HTML, are certainly the most obvious example of the role that standardisation can play in achieving a significant level of interoperability. However, the evolution of the Internet into what is called the "Future Internet" challenges existing standards [PSDZ12]. Internet standards only ensure that components on different computers can pass data over, but do not guarantee that the components exchange *meaningful information* as stressed by Definition 3. In this direction, the Semantic Web [BLHL+01] promotes the vision in which Web resources are augmented with machine-processable metadata expressing their meaning. This vision is supported by *ontologies*, which provide a machine-processable means to represent and automatically reason about the meaning of data based on the shared understanding of the domain [Gru09].

Another downside of standards is that there are so many to choose from [TW10, pp.702]. For example, one can use FTP or WebDAV, both of which are standards, for file sharing, but a user of one standard cannot share files with a user of the other standard, unless there is some intermediary software that enables them to work together. In conclusion, there is no doubt about the importance of standards to facilitate interoperability but they are neither necessary nor sufficient to guarantee interoperability [LMSW08].

Let us now consider standards from a broader perspective, just as a form of agreement between parties [TMD09]. Due to the increasing ubiquity of modern software systems and the high demand on runtime support, such an agreement is often impossible to reach. Many connected devices such as smartphones, tablets, and laptops are an integral part of our lives and assist us in a wide range of daily tasks. But to take full advantage of the digital ecosystem, the software components within these devices should be able to discover the services around them dynamically and interact with them at runtime. It is impossible to predict all possible interactions, and hence impossible to agree on the rules that would enable them to work together. Reaching a common agreement may also be impossible due to the complexity of the components involved, which can be systems in themselves. This is the case of inter-

operability in the context of systems of systems [MLM+04]. Examples include the military systems of different countries [Lar03], or several transportation companies within a city [SEC12]. The components of systems of systems are completely autonomous systems, which are designed and implemented independently and do not obey any central control or administration. Nonetheless, there are real incentives for these components to work together, e.g., to allow international cooperation during conflicts, or ensure users can commute. In this context, it is not enough to simply assess the ability of the components to work together but rather we should *make* them work together.

In the light of the above observations, we revise the definition of interoperability as follows:

Definition (Interoperability)**.** *Interoperability characterises the extent to which two software components from different manufacturers, which are functionally compatible, can be made to work together correctly by reconciling the differences in their interfaces and behaviours.*

In order to make functionally-compatible components work together, without modifying them, intermediary software entities, called *mediators*, are used [Wie92]. Mediators achieve interoperability by reconciling the differences in the implementations of the components involved. Designing and implementing mediators requires dealing with many concerns: (i) coordination of the behaviours of the components so as to guarantee their correct interaction (e.g., absence of deadlocks), (ii) data translation so as to ensure meaningful information exchange between the components, and in the case of distributed components (iii) communication between the components so as to address the issues inherent in their distribution across the network (e.g., concurrency and fault tolerance).

The enabling technology that facilitates the development of mediators by addressing the issues inherent in the distribution of components is *middleware*. Middleware is a software entity logically placed between the application and the operating system that provides an abstraction that facilitates the communication and coordination of distributed components [TVS06]. Middleware provides mediation at the infrastructure layer as it makes components work together by hiding the differences in hardware and operating systems. 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, there is the problem of differences between middleware technologies themselves. For example, a CORBA client cannot access a SOAP-based Web Service. Still, while SOAP and CORBA respect a similar pattern for interaction —typically a client sends a request to the service, which processes it and returns the corresponding response to the client— there are other examples of middleware that rely on a fundamentally different type of interaction, e.g., JavaSpaces whose components interact via a shared memory. Hence, not only do current middleware solutions not fully solve the interoperability problem, but they may even aggravate it by adding differences at the middleware layer.

Furthermore, even applications developed using the same middleware are not guaranteed to work together so long as there are differences in their interfaces and behaviours. This is, for example, the case of interoperability in Web Services [NBCT06]. Even though both services and clients use SOAP middleware, the differences between their interfaces, which include differences in the operation names, input/output message names and types, the granularity of operations, and the order in which these operations are invoked (or expected to be invoked) hamper independently-developed clients and Web Services from working together. A mediator is required to enable them to interoperate. Given the countless number of potential cases where a mediator is necessary, any static solution is doomed to fail. We need to generate mediators automatically.

The ultimate goal of this thesis work is to achieve interoperability between functionally-compatible components by automatically synthesising mediators that reconcile the differences between the implementations of these components. Our thesis statement is the following:

> *"Components with compatible functionalities should be able to interoperate despite differences in their respective implementations. By reasoning about the meaning of the information they exchange and analysing their behaviours, we can automatically synthesise and dynamically deploy mediators so as to achieve interoperability between these components. The mediators solve the differences between the interfaces of the components and coordinate their behaviours at both the application and middleware layers."*

Our approach for the automated synthesis of mediators relies on the adequate specifications of the components, which describe the meaning of the actions of the components' interfaces together with the components' behaviours. Once deployed, the

mediators reconcile the differences between component implementations at both application and middleware layers. The systematic approach for generating mediators makes our solution applicable at both design time and runtime, and suitable for future and unforeseen components. Furthermore, since a solution for achieving interoperability can only be useful if it can be successfully applied to complex, real-world problems, we investigate many case studies where interoperability is a major concern and show how it can be efficiently solved using the automated synthesis of mediators.

1.2 Case Studies

The need for interoperability is encountered in many domains such as instant messaging, collaborative file management, e-business, event management, and emergency management. We present the major use cases that we investigated during this thesis in order of increasing complexity ranging from the simplest case, where the mediator just translates each message sent by one of the components into a message that the other component expects, to the most complex case, which involves a system of systems for the management of emergency situations. By considering several case studies, we gauge the problems frequently encountered when achieving interoperability and validate the assumptions that can be made.

The first case study relates to Instant Messaging (IM). IM is a popular application for many Internet users —over 3.1 billion users in 2012 and over 3.8 billion users predicted for 2016 [Rad12]— with an increasing emphasis on mobility —30% of smartphones had IM applications installed in 2012 [Rad12]. However, users of one IM application cannot exchange messages with the users of another application. Although this situation may be frustrating from a user perspective, it reflects the way IM, like many other applications, has evolved. There exist many solutions that integrate the disparate applications but they often require developing and maintaining mediators manually. Even though these mediators are simple —often just intercepting the messages from one client application and translating them into messages of the interacting application— they are impractical in the long term as new versions and standards keep emerging. However, the existence of these solutions, and their widespread use, demonstrate the need for interoperability in the IM domain. Yet, despite this need for interoperability and despite the existence of a W3C standard,

XMPP[1], the situation seems unlikely to change.

We move to the case of interoperability between file management systems. Users may use different applications to organise, search, and manipulate their files, and share them with other users. We concentrate on two systems: WebDAV[2], which is an IETF standard, and Google Docs[3]. Although these two systems offer similar functionalities and use HTTP as the underlying transport protocol, they are unable to interoperate. For example, a user cannot access his Google Docs documents using his favourite WebDAV client (e.g., Mac Finder). This is mainly due to the syntactic naming of data and operations used in each system, and the order in which these operations are performed. This case study also illustrates the case of one-to-many correspondence between the actions of the two components, i.e., one action required from a component is translated into a sequence of actions provided by the other component.

We also consider the Purchase Order scenario proposed as part of the Semantic Web Service (SWS) Challenge [PMLZ08]. The scenario represents a typical real-world problem that is close to industrial reality. It is intended as a common ground to discuss semantic (and other) Web Service solutions and compare them according to the set of features that a mediation approach should support.

We move on to the case of differences spanning both the application and middleware layers. We consider the case of interoperability between event management systems, which are concerned with the organisation of events like conferences, seminars, and concerts. We concentrate on the case of Amiando[4] and RegOnline[5], which were built as REST and SOAP Web Services respectively. They further exhibit differences in the name and granularity of the operations they propose.

Finally, to highlight the interoperability challenge in systems of systems, we consider one representative application domain, that of global monitoring of the natural environment, as illustrated by the GMES[6] initiative, which we investigated in the context of the CONNECT European Project [Con12]. GMES is the European Programme for the establishment of a European capacity for Earth Observation. Special interest

[1] Extensible Messaging and Presence Protocol – http://www.xmpp.org/
[2] Web Distributed Authoring and Versioning – http://www.ietf.org/rfc/rfc2518.txt
[3] http://docs.google.com
[4] http://developers.amiando.com/
[5] http://developer.regonline.com/
[6] Global Monitoring for Environment and Security – http://www.gmes.info/

is given to the support of emergency situations across different European countries. In emergency situations, the context is highly dynamic and involves highly heterogeneous components that interact in order to perform the different tasks necessary for decision making. The tasks include, among others, collecting weather information, and capturing video using different devices. GMES makes a strong case for the need for on-the-fly solutions to interoperability in systems of systems. Indeed, each country defines an emergency management system that encompasses different components, which interact according to standards specific to the country. However, in special circumstances, assistance may come from other countries, which bring their own components defined using different standards. Besides demonstrating the need for mediation at runtime, GMES also requires achieving interoperability between middleware with different interaction patterns, namely remote procedure call and publish/subscribe.

We use these case studies throughout the document to illustrate our approach and also to validate and assess the solution in a dedicated chapter.

1.3 Influences

Interoperability has received a great deal of interest and led to the provision of a multitude of solutions, both theoretical and practical, albeit primarily oriented toward design time. The approach for the automated synthesis and deployment of mediators we propose in this thesis takes its inspiration from the extensive work on interoperability seen from the perspective of its underpinning fields: software architecture, middleware, formal methods and Semantic Web. Figure 1.1 depicts, for each perspective, the specific focus and the main technique for supporting interoperability as well as how mediators are considered.

- *Software architecture.* Software architecture focuses on *composition*: several software entities are put together to build a system and define its *structure* as a whole [Sha93]. Interaction between components is abstractly described using software connectors. In other words, connectors model the exchange of information between components and the coordination of their behaviours. Hence, mediators can be conveniently represented as connectors. One critical issue for software architecture is the design and implementation of the connectors

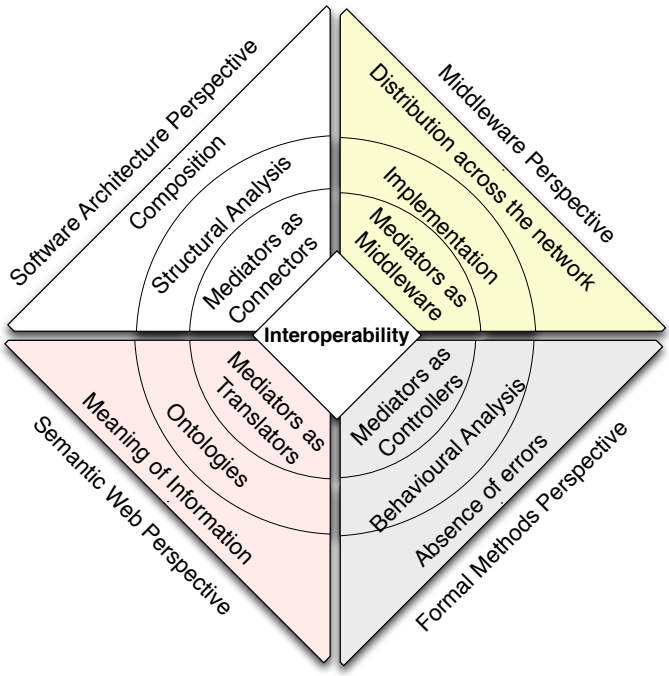

Figure 1.1. The different perspectives on interoperability

that permit the various software components to work together properly. While middleware offers convenient services that facilitate the implementation of connectors, the design and specification of mediators remain the responsibility of developers.

- *Middleware.* Middleware provides an abstraction that facilitates the development of *distributed* applications despite the heterogeneity of the underlying infrastructure [TVS06], as depicted in Figure 1.2. On the one hand, middleware makes applications agnostic to the differences between the operating systems and the hardware of the devices on which software components are deployed. On the other hand, applications implemented using different middleware solutions are not able to work together.

  Therefore, other middleware solutions have been proposed in order to reconcile the differences between middleware. In this case, the proposed middleware acts as a mediator between different middleware implementations. In the simple case of middleware obeying the same interaction pattern, the differences can simply be solved by translating the messages sent using one middleware into suitable

messages expected by the other middleware. However, when the middleware implementations follow different interaction patterns, e.g., shared memory and publish/subscribe, then the differences are such that they cannot always be solved [CMP08]. What is needed is to further consider the characteristics of the applications so as to verify whether the differences can indeed be reconciled.

Considering the innumerable cases of potential applications, defining some middleware that ensures interoperability by considering the application characteristics and associated middleware is unfeasible. It is necessary to define solutions able to reason about the characteristics of applications automatically in order to infer the necessary actions for reconciling the differences between component implementations. As sketched below, formal methods focus on reconciling the behaviours of the components while Semantic Web technologies focus on the meaning of the information exchanged between components.

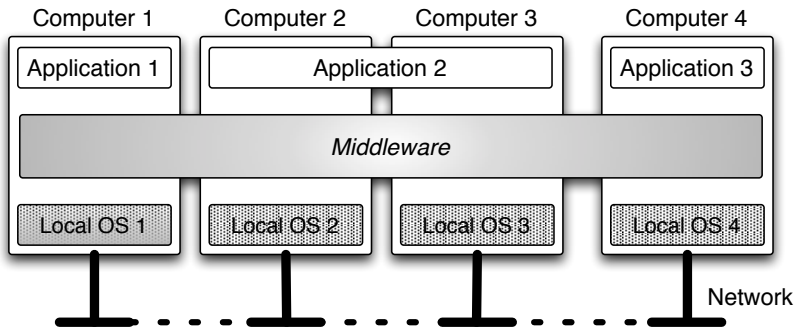

Figure 1.2. Middleware

- *Formal methods.* Formal methods are mathematically-based languages, techniques, and tools for specifying and verifying hardware and software systems [CW96]. Formal methods focus on the *behaviour* of software systems, which they rigorously analyse in order to reveal potential inconsistencies, ambiguities, and incompleteness. In other words, formal methods help to verify the *absence of execution errors* in software systems. Once potential execution errors (a.k.a. mismatches) are detected, they can be solved either by eliminating the interactions leading to the errors or by introducing a controller that forces the components to coordinate their behaviours correctly. Only the introduction of a controller can keep the functionality of the system intact by enabling its components to achieve their individual functionalities.

Existing solutions for the generation of controllers (e.g., [YS97, BBC05, Sal10, MPS12]) often operate on a high-level abstraction, which makes turning the generated controller into an implementation very challenging. Moreover, they either assume that the behaviours of the functionally-compatible components are described using the same set of actions, a specification of the composed system is given, or the correspondence between the actions of components' interfaces is provided [DG09]. This assumption does not hold in the case of today's ubiquitous computing environments, where interactions between components can only be known at runtime [CK10].

- *Semantic Web.* The Semantic Web is an extension of the Web in which information is given well-defined *meaning*, better enabling computers and people to work in cooperation [BLHL+01]. Ontologies play a key role in the Semantic Web by formally representing shared knowledge about a domain of discourse as a set of concepts, and the relationships between these concepts [Gru93]. Ontologies have been extensively used to automate the reasoning about the information exchanged between software components, especially in ubiquitous computing environments, so as to infer the translations necessary to reconcile the differences in the syntax of this information [MSZ01]. However, these translations solve interoperability only at the application layer.

To sum up, software architecture provides us with techniques to understand interoperability and reason about the composition of software components but does not specify how to achieve interoperability automatically. Middleware gives us reusable solutions that facilitate the implementation of mediators but these solutions do not reconcile the differences between components at the application layer. Formal methods provide us with the foundations for coordinating the behaviours of components in order to guarantee the absence of errors in their interactions, but assume that either the components use the same set of actions, a specification of the composed system is given, or the correspondence between their actions is provided. Finally, Semantic Web technologies allow us to infer the translations necessary to ensure meaningful exchange of information between components, but do not deal with the differences between components at the middleware layer.

1.4 Contributions

We posit that interoperability should neither be achieved by defining yet another middleware nor yet another ontology but rather by exploiting existing middleware together with knowledge encoded in existing domain ontologies to synthesise and implement mediators automatically. We define a multifaceted approach to interoperability, which brings together and enhances the solutions that tackle interoperability from different perspectives, as depicted in Figure 1.3. The mediators we synthesise act as (i) translators by ensuring the meaningful exchange of information between components, (ii) controllers by coordinating the behaviours of the components to ensure the absence of errors in their interaction, and (iii) middleware by enabling the interaction of components across the network so that each component receives the data it expects at the right moment and in the right format.

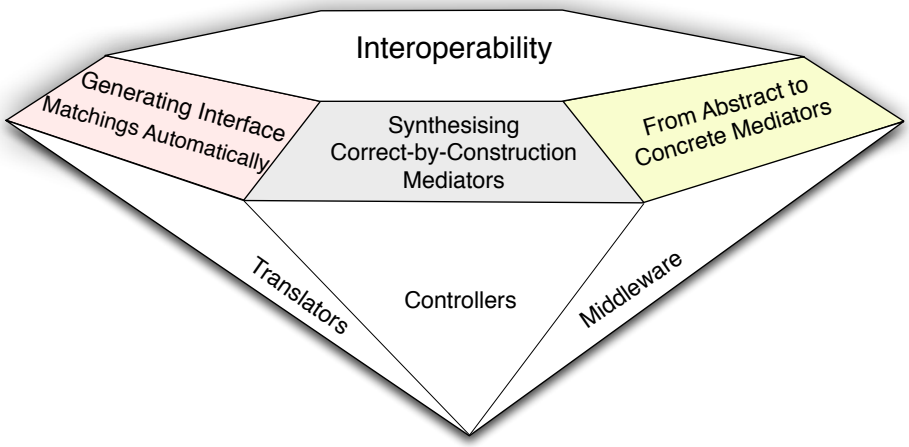

Figure 1.3. A multifaceted approach for interoperability

The inputs of our approach are two functionally-compatible components, which are described using their interfaces and behaviours together with a domain ontology that represents the shared knowledge of the application domain. In order to enable interoperability between functionally-compatible components, we make the following contributions:

- *Generating interface matchings automatically.* A significant role of the mediator is to translate information available on one side and make it suitable and relevant to the other. This translation can only be carried out if there exists a

semantic correspondence between the actions required by one component and those provided by the other component, that is, *interface matching*. The main idea is to use the domain-specific information embodied in the domain ontology in order to select from sequences of actions of the components' interfaces only those which retain the meaning of the information exchanged and for which translations can automatically be computed. The generated interface matchings not only specify one-to-one correspondences between the actions of components but also many-to-many correspondences, which makes their computation very complex. By using constraint programming, which we leverage to support ontology reasoning, we are able to calculate interface matchings efficiently.

- *Synthesising correct-by-construction mediators.* We explore the behaviours of the functionally-compatible components in order to generate the mediator that composes the computed interface matchings to guarantee the correct interaction between the components. The algorithm deals with the potential ambiguity of interface matching, i.e., when the same sequence of actions of one component matches with different sequences of actions from the other component. The mediator, if it exists, guarantees that the two components progress synchronously and reach their final states.

- *From abstract to concrete mediators.* Ensuring interoperability from application down to middleware often requires dealing with many concerns, some of which are specific to the application at hand while the others relate to the distribution and coordination of the components. We perform the automated synthesis of mediators at the ontological and behavioural levels where reasoning and inference can be realised. Then, we refine the mediators by considering the interaction patterns of the middleware involved. Finally, we use middleware libraries to create concrete network messages based on these middleware primitives. Only when the mediator includes all the details about the communication of components, can interoperability be achieved.

- *Experimenting with real-world cases.* We demonstrate the validity of our approach through the development of the MICS (Mediator Synthesis to Connect Components) tool and illustrate its large applicability using real-world scenarios involving heterogeneous existing systems. We show that our approach is

able to automatically generate the appropriate mediators automatically. The mediator enables the components to interoperate while introducing only a small overhead.

Much of this work has been either published or submitted for publication, but this thesis should be regarded as the definitive and uniform exposition of the approach.

- In [IBB11, IB13], we review literature on interoperability and present the foundation for the automated synthesis of mediators to achieve interoperability from application down to middleware.
- In [BBG+11], we focus on the role of ontologies to support interoperability in distributed systems.
- In [BBG+13], we present the approach from the perspective of models at runtime.
- In [BRA+11, BIR+11, BIS+12], we explore the role of learning techniques to complete the model of software components based only on their syntactic interfaces.
- In [BCI+13], we investigate the use of quotient and ontology reasoning for the of synthesis of correct-by-construction mediators.
- In [BI13], we introduce the automated generation of interface matching using constraint programming and the related synthesis of correct-by-construction mediators.
- In [BIST12], we present the results of using automated synthesis of mediators to enable interoperability across different instant messaging applications.

1.5 Thesis Outline

This thesis is organised as follows:

- Chapter 2 surveys existing approaches to reason about and achieve interoperability from its four underpinning perspectives: (i) software architecture to understand and reason about interoperability, (ii) middleware to investigate

methods for the implementation of mediators, (iii) formal methods to analyse the behaviours of the components and synthesise mediators, and (iv) Semantic Web technologies to represent and reason about domain knowledge and ensure meaningful data exchange between components.

- Chapter 3 introduces the context in which the approach is the most relevant. It explains the foundations of the CONNECT approach for achieving interoperability in today's and future software systems by discovering the components spread over the network, completing their specifications, and synthesising the appropriate mediators that enable them to interoperate.

- Chapter 4 details the approach for the automated synthesis of mediators. It presents a solution based on ontology reasoning and constraint programming so as to compute interface matchings efficiently. It also describes the algorithm used to generate the mediator that guarantees that the components interact successfully.

- Chapter 5 considers the concretisation of the synthesised mediator further considering heterogeneous middleware implementations, including those based on different interaction patterns.

- Chapter 6 presents the MICS tool that is the implementation of our approach for the automated synthesis of mediators. It also demonstrates the viability and wide applicability of the approach through different case studies.

- Chapter 7 emphasises the contribution of this work and identifies challenges and research gaps that require further exploration.

Chapter 2

Interoperability: A Landscape of the Research Field

"The world is moved along, not only by the mighty shoves of its heroes, but also by the aggregate of the tiny pushes of each honest worker."

— Helen A. Keller, lecturer and author (1880-1968)

Ask a software architect about interoperability, and he will answer that it is about the development of the software connector that enables components to interact successfully. Ask a middleware developer and he will tell you that it is about defining an appropriate middleware. Ask a formal methods expert and he will say that it is about computing a controller that forces the components to interact without errors. Ask a Semantic Web expert and he will tell you that it is about defining an ontology that enables reasoning about the meaning of the information exchange. In this chapter, we review the literature on interoperability from these four perspectives. We first adopt a software architecture perspective to present the concepts underpinning interoperability. Next, we concentrate on middleware for the implementation of concrete software solutions to achieve interoperability. Then, we describe formal solutions that analyse the behaviours of components in order to synthesise the mediator that guarantees that they can interact without errors. Finally, we present solutions based on ontologies so as to represent and reason about the meaning of the information exchanged between components at runtime, as is required by ubiquitous computing.

To illustrate existing solutions and show their benefits and limitations, we use a simple example from the GMES case study (see Figure 2.2). In *Country 1*, the Command and Control centre (*C2*) obtains weather information by interacting with *Weather Service* using SOAP. In *Country 2*, *Weather Station* provides specific information such as temperature or humidity and is implemented using CORBA. *C2* requires a weather functionality that can be provided by *Weather Station*, however, they cannot interact successfully without a mediator that reconciles the differences in their implementations.

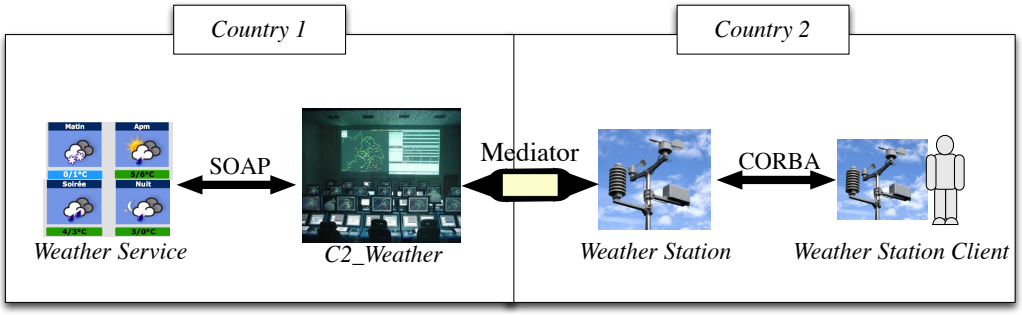

Figure 2.1. The weather example

The successful interoperation of components is often hampered by differences in their implementations at both the middleware and application layers. At the application layer, components may define different data types and operations, and have divergent behaviours. At the middleware layer, components may rely on various middleware technologies (e.g., SOAP, CORBA, or JMS) that use distinct data representation formats, and define disparate interaction patterns. As the differences between components span both the application and the middleware layers, interoperability becomes a cross-cutting concern and must be achieved from the application down to the middleware, while we consider that differences between operating systems and network protocols are handled by the middleware.

Figure 2.2 illustrates a classification of the differences between the components that can prevent them from interoperating in the case of the weather example. At the application layer, *C2* uses the Weather data type whereas *Weather Station* defines the Temperature and Humidity data types. *C2* performs a single action getWeather while *Weather Station* provides two actions getTemperature and getHumidity, which can be performed independently. At the middleware layer, *C2* sends a SOAP request

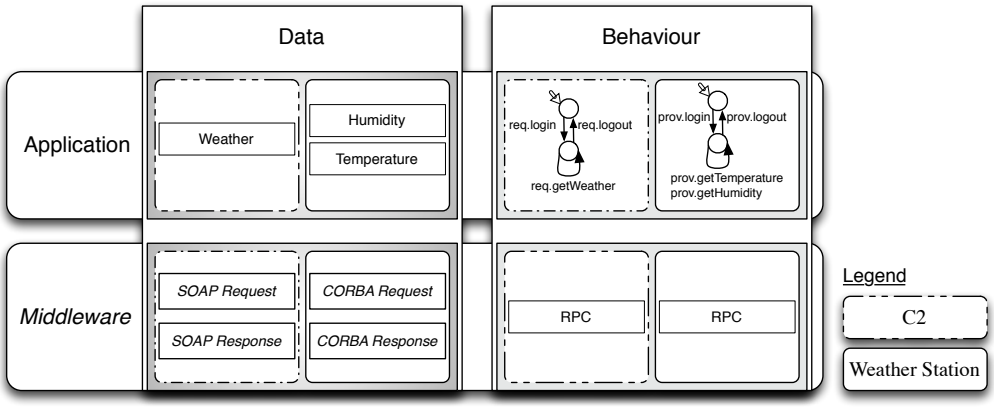

Figure 2.2. Interoperability barriers

to invoke an action and expects a SOAP response in return, whereas *Weather Station* expects CORBA requests, and sends the results in CORBA responses. Consequently, *C2* is unable to invoke the actions of *Weather Station* even though the interaction pattern is the same, i.e., Remote Procedure Call (RPC).

This classification of interoperability barriers shows that differences between components span both the application and middleware layers, and concern both the data manipulated by components and the behaviours of these components. This classification is by no means exhaustive as our aim is not to introduce yet another classification of interoperability barriers —plenty have already been proposed [BOR04, UCJ09]— but rather to set up a framework that will help us understand, discuss, and compare the different approaches to interoperability.

2.1 The Software Architecture Perspective: Understanding Interoperability

Software architecture abstractly describes the structure of software systems in terms of components and connectors: *components* are meant to encapsulate computation while *connectors* are meant to encapsulate interaction [Sha93].

In order to facilitate software composition and reuse, a component encapsulates some functionality to which it restricts access via an explicit interface [TMD09]. The *interface of a component* specifies the set of observable *actions* that the component uses to interact within its running environment in order to perform its functionality.

This set is partitioned into *required* and *provided* actions, with the understanding that required actions are received from and controlled by the environment, whereas provided actions are emitted and controlled by the component.

The architectural element tasked with effecting and regulating interactions between components is a connector [TMD09]. The implementation of a connector is often based on middleware [MDT03] since middleware provides reusable solutions that facilitate communication and coordination between components. For this purpose, middleware defines [ICG07]: (i) an Interface Description Language (IDL) for specifying the interfaces of components and the associated operations, and data types, (ii) a discovery protocol to address and locate the components that are available in the environment, (iii) an interaction protocol that coordinates the behaviour of different components and enables them to collaborate, and (iv) additional protocols to manage non-functional requirements such as dependability, security, fault-tolerance, and performance optimisation. However, while components and connectors are conceptually separate, the middleware used to implement a connector is often invasive in that it influences the implementation of the components. For example, the language used to describe the component's interface differs according to the middleware implementation used, e.g., CORBA IDL in the case of CORBA-based middleware and WSDL in the case of SOAP-based middleware.

Figure 2.3 depicts the interaction between *C2* and *Weather Service*. The interface of *C2* includes three required actions *req.login*, *req.getWeather*, and *req.logout*. The interface of *Weather Service* encompasses three provided actions *prov.login*, *prov.getWeather*, and *prov.logout*. The connector *Weather_Connector1* models the interaction between *C2* and *Weather Service*. *Weather_Connector1* is implemented using SOAP, which also implies that *C2* acts as a SOAP client when performing its

Figure 2.3. *C2*, *Weather Service*, and associated connector in *Country 1*

required actions and *Weather Service* acts as a SOAP service when providing these same actions.

Similarly, Figure 2.4 depicts the interaction between the *Weather Station* component and its specific client. The interface of *Weather Station* specifies four provided actions *prov.login*, *prov.getTemperature*, *prov.getHumidity*, and *prov.logout*. *Weather Station Client* exhibits the dual interface with required actions. The interaction between *Weather Station* and *Weather Station Client* is abstracted using *Weather_ Connector2*. The use of CORBA to implement *Weather_ Connector2* means that *Weather Station Client* carries out its required actions by sending CORBA requests and receiving CORBA responses while *Weather Station* provides its actions by receiving CORBA requests and sending CORBA responses.

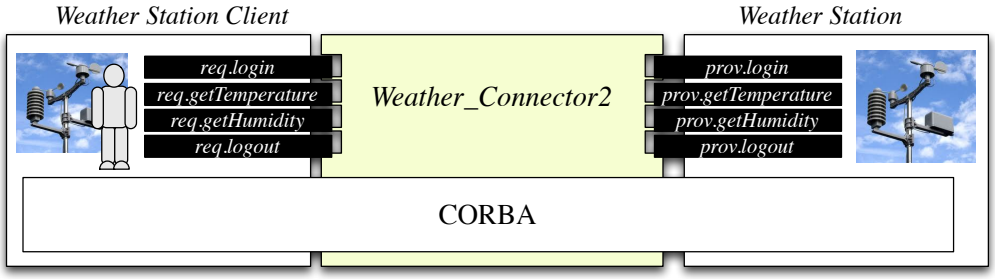

Figure 2.4. *Weather Station*, its client, and associated connector in *Country 2*

Let us now consider the case of *C2* willing to interact with *Weather Station*. The former requires a weather functionality that is provided by the latter. However, neither *Weather_ Connector1*, nor *Weather_ Connector2* can readily be used to manage their interaction. Intuitively, we can state that *C2* and *Weather Station* cannot operate together. This is known as an *architectural mismatch*. Architectural mismatches occur when composing two, or more, software components to form a system and those components make conflicting assumptions about their environment [GAO95], thereby preventing interoperability. These assumptions relate to: (i) the interfaces and behaviours of the components involved, (ii) the behaviours and implementations of the connectors used, and (iii) the operating systems and the hardware of the devices on top of which the components are deployed.

Architectural mismatches represent barriers to interoperability that must be solved in order to enable functionally-compatible components to work together. To achieve interoperability automatically, this intuitive notion of architectural mismatches must

be made more precise. First, we present the formal foundation of software architecture so as to allow us to reason about interoperability. Then, we introduce mediators as a means to achieve interoperability. Finally, we discuss the synthesis of mediators at runtime in order to enable interoperability in highly-dynamic environments, including ubiquitous computing environments.

2.1.1 Formal Reasoning about Interoperability

The first step towards reasoning about interoperability is by formalising component interactions. The *behaviour of a component* specifies its interaction with the environment and models how the actions of its interface are coordinated to achieve its functionality.
Different languages may be considered for the specification of a component's behaviour. Formal languages are a prerequisite for automated analysis of the component's behaviour while standard, well-established languages (e.g., BPEL[1] and CDL[2]) are easier for developers to deal with. We build upon pioneering work by Allen and Garlan [AG97], which uses process algebra to model the behaviours of components together with their interaction. More specifically, we use FSP (Finite State Processes) [MK06] based on the follow-up work by Spitznagel and Garlan, which in particular considers the transformation of connectors in order to address dependability as well as interoperability concerns [SG03]. It is worth noticing that there further exists a tool, WS-Engineer [Fos08], to convert BPEL and CDL specifications into FSP descriptions automatically. In this section, we present FSP and show how it is used to model interactions between components and reason about interoperability.

2.1.1.1 Finite State Processes

FSP [MK06] is a process algebra that has proven to be a convenient formalism for specifying concurrent components, analysing, and reasoning about their behaviours. Briefly stated, FSP processes describe actions (events) that occur in sequence, and choices between action sequences. Each process has an alphabet, αP, of the actions that it is aware of (and either accepts or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are con-

[1] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[2] http://www.w3.org/TR/ws-cdl-10/

structed through action prefix $a \rightarrow P$, choice $a \rightarrow P|b \rightarrow Q$, and sequential composition $P;Q$. Composite processes are constructed using parallel composition $P\|Q$ or process relabelling $a : P$. The replicator forall is a convenient syntactic construct used to specify parallel composition over a set of processes. Processes can optionally be parameterised $P(X =' a)$ and have re-labelling $P/\{new\_1/old\_1, ..., new\_n/old\_n\}$, hiding $P\backslash\{a_1, ..., a_n\}$ or extension $P + \{a_1, ..., a_n\}$ over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with $\|$.

The semantics of FSP are given in terms of Labelled Transition Systems (LTS) [Kel76]. The LTS interpreting an FSP process P can be regarded as a directed graph whose nodes represent the process states and each edge is labelled with an action $a \in \alpha P$ representing the behaviour of P after it engages in a. $P \xrightarrow{a} P'$ then denotes that P transits with action a into P', and $P \stackrel{s}{\Rightarrow} P'$ is shorthand for $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 ... \xrightarrow{a_n} P'$ where $s = \langle a_1, a_2, ..., a_n \rangle, a_{i=1..n} \in \alpha P$. There exists a start node from which the process begins its execution. The END state indicates a successful termination. When composed in parallel, processes synchronise on shared actions: if processes P and Q are composed in parallel, actions that are in the alphabet of only one of the two processes can occur independently, but an action that is in the alphabets of both cannot occur until both processes are willing to engage in it.

We provide an overview of the syntax and semantics of FSP operators in Appendix A and refer the interested reader to [MK06] for further details.

2.1.1.2 Formalising Components and Connectors using FSP

Based on pioneering work for formalising component interactions in software architecture [AG97], the behaviour of a component is modelled using *ports* while a connector is modelled as a set of *roles* and a *glue*. The roles specify the expected behaviours of the interacting components while the glue describes how the behaviours of these components are coordinated. More specifically, we consider the specification of the ports, roles, and glue as FSP processes.

To return to the example in Figure 2.4. To achieve the required weather functionality, *C2* first logs in, invokes *getWeather* several times, and finally logs out. We specify this behaviour as follows:

$$
\begin{array}{lcl}
C2\_weather\_port & = & (req.login \rightarrow P1), \\
P1 & = & (req.getWeather \rightarrow P1 \mid req.logout \rightarrow C2\_weather\_port).
\end{array}
$$

Since *C2* interacts using SOAP, then each of the required actions is realised by invoking the appropriate operation *op*, which belongs to the set $\{login, getWeather, logout\}$, by sending a SOAP request and receiving a SOAP response, which is formalised as follows:

$$SOAPClient\ (X =' op) = (req.[X] \rightarrow sendSOAPRequest[X] \rightarrow receiveSOAPResponse[X] \rightarrow SOAPClient).$$

The *Weather_ Connector1* connector managing the interactions between *C2* and *Weather Service* defines a role associated with each of them, that is, *C2_ weather_ role* and *WeatherService_ role*, respectively. The connector also defines how these actions are realised using SOAP. More specifically, the *SOAPClient* process specifies that each required action corresponds to the sending of a SOAP request, which includes the name of the SOAP operation as a parameter, and the reception of the corresponding SOAP response. Dually, the *SOAPServer* process specifies that each provided action corresponds to the reception of a SOAP request, which includes the name of the SOAP operation as a parameter, and the sending of the corresponding SOAP response. The *SOAPGlue* process specifies that for each operation belonging to the set $\{login, getWeather, logout\}$, a request sent by the SOAP client is received by the SOAP server, and then the response sent by the SOAP server is received by the SOAP client. *Weather_ Connector1* is specified as the parallel composition of all these processes:

$$\begin{aligned}
&\mathsf{set}\ weather_actions1 = \{login, getWeather, logout\}\\
&C2_weather_role = (req.login \rightarrow P1),\\
&P1 = (req.getWeather \rightarrow P1 \mid req.logout \rightarrow C2_weather_role).\\
&WeatherService_role = (prov.login \rightarrow P2),\\
&P2 = (\ prov.getWeather \rightarrow P2 \mid prov.logout \rightarrow WeatherService_role).\\
&SOAPClient\ (X =' op) = (req.[X] \rightarrow sendSOAPRequest[X] \rightarrow receiveSOAPResponse[X] \rightarrow SOAPClient).\\
&SOAPServer\ (X =' op) = (prov.[X] \rightarrow receiveSOAPRequest[X] \rightarrow sendSOAPResponse[X] \rightarrow SOAPServer).\\
&SOAPGlue\ (X =' op) = (sendSOAPRequest[X] \rightarrow receiveSOAPRequest[X] \rightarrow sendSOAPResponse[X] \rightarrow receiveSOAPResponse[X] \rightarrow SOAPGlue).\\
&\ldots
\end{aligned}$$

$$
\begin{array}{l}
\ldots \\
\| \mathit{Weather\_Connector1} \;=\; (\; \mathit{C2\_weather\_role} \\
\qquad\qquad\qquad\qquad \| \quad \mathit{WeatherService\_role} \\
\qquad\qquad\qquad\qquad \| \quad (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{SOAPClient}(\mathrm{op})) \\
\qquad\qquad\qquad\qquad \| \quad (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{SOAPGlue}(\mathrm{op})) \\
\qquad\qquad\qquad\qquad \| \quad (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{SOAPServer}(\mathrm{op}))).
\end{array}
$$

Let us now consider the example in Figure 2.4. *Weather Station* expects clients to login first, then ask for the temperature or humidity several times, and log out to terminate. Note that the two actions $prov.getTemperature$ and $prov.getHumidity$ are performed independently. For each provided action, *Weather Station* receives a CORBA request, which it processes, and then sends the corresponding response. The port of *Weather Station* is specified as follows:

$$
\begin{array}{lcl}
\mathit{WeatherStation\_port} & = & (prov.login \rightarrow P2), \\
P2 & = & (\; prov.getTemperature \rightarrow P2 \\
 & | & \quad prov.getHumidity \rightarrow P2 \\
 & | & \quad prov.logout \rightarrow \mathit{WeatherStation\_port}). \\
\mathit{CORBAServer}\ (X =' op) & = & (prov.[X] \rightarrow receiveCORBARequest[X] \\
 & & \rightarrow sendCORBAResponse[X] \rightarrow \mathit{CORBAServer}).
\end{array}
$$

The *Weather_ Connector2* connector models the interaction between *Weather Station* and the corresponding client. It is implemented using CORBA. *Weather_ Connector2* is specified as follows:

$$
\begin{array}{lcl}
\mathsf{set}\ \mathit{weather\_actions2} & = & \{login, getTemperature, getHumidity, logout\} \\
\mathit{WeatherStationClient\_role} & = & (req.login \rightarrow P1), \\
P1 & = & (\; req.getTemperature \rightarrow P1 \\
 & | & \quad req.getHumidity \rightarrow P1 \\
 & | & \quad req.logout \rightarrow \mathit{WeatherStationClient\_role}). \\
\mathit{WeatherStation\_role} & = & (prov.login \rightarrow P2), \\
P2 & = & (\; prov.getTemperature \rightarrow P2 \\
 & | & \quad prov.getHumidity \rightarrow P2 \\
 & | & \quad prov.logout \rightarrow \mathit{WeatherStation\_role}). \\
\mathit{CORBAClient}\ (X =' op) & = & (req.[X] \rightarrow sendCORBARequest[X] \\
 & & \rightarrow receiveCORBAResponse[X] \rightarrow \mathit{CORBAClient}). \\
\mathit{CORBAServer}\ (X =' op) & = & (prov.[X] \rightarrow receiveCORBARequest[X] \\
 & & \rightarrow sendCORBAResponse[X] \rightarrow \mathit{CORBAServer}). \\
\ldots
\end{array}
$$

$$
\begin{array}{lll}
\ldots & & \\
CORBAGlue\ (X =' op) & = & (sendCORBARequest[X] \rightarrow receiveCORBARequest[X] \\
 & & \rightarrow sendCORBAResponse[X] \rightarrow receiveCORBAResponse[X] \\
 & & \rightarrow CORBAGlue). \\
\| Weather\_Connector2 & = & (\ WeatherStationClient\_role \\
 & & \|\ \ \ WeatherStation\_role \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]CORBAClient(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]CORBAGlue(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op}))).
\end{array}
$$

2.1.1.3 Reasoning about Architectural Mismatches

A component can be attached to a connector only if its port is *behaviourally compatible* with the connector role it is bound to. Behavioural compatibility between a component port and a connector role is based upon the notion of refinement, i.e., a component port is behaviourally compatible with a connector role if the process specifying the behaviour of the former refines the process characterising the latter [AG97]. This refinement implies the inclusion of the traces of the expected behaviour of the component in those of the observed behaviour of the component. In other words, it should be possible to substitute the role process by the port process. Refinement gives an intuitive notion of correctness (at least for safety properties), and it has been applied in the stepwise design and implementation of software systems, starting from their more abstract specification. Such reasoning exploits the expressive and deductive power of the mathematics of sets and sequences [Hoa04].

For example, *C2* can be attached to *Weather_Connector1* since *C2_weather_port* refines *C2_weather_role* —they are actually defined the same way. Likewise, *WeatherStation_port* refines *WeatherStation_role* defined by *Weather_Connector2*. However, *WeatherStation_port* cannot be attached to *Weather_Connector1* since it does not refine any of its ports, nor can *C2_weather_role* be attached to *Weather_Connector2*. Hence, in the case of *C2* willing to interact with *Weather Station*, none of the available connectors can directly be used, resulting in an architectural mismatch.

Verifying behavioural compatibility between components' ports and connectors' roles allows us to check the presence or absence of architectural mismatches and also to suggest a solution. To solve architectural mismatches, we must find or create a connector whose roles are refined by components' ports. Such a connector is called a mediator.

2.1.2 Mediators to Support Interoperability

To enable interoperability between functionally-compatible components, the mediator must solve architectural mismatches by reconciling the conflicting assumptions that the components make about their environment. We recall that these assumptions relate to: (i) the interfaces and behaviours of the components involved, (ii) the behaviours and implementations of the connectors used, and (iii) the operating systems and the hardware of the devices on top of which the components are deployed. To solve the differences between the interfaces of components, the mediator must translate the actions required by each of them into actions provided by the other. Note that the mediator facilitates interaction —it is a connector— but does not provide any action itself since it does not encapsulate computation. To solve the differences between the behaviours of components, the mediator must coordinate the exchange of information between these components by controlling which action should be delivered to which component at which time. To solve the differences between the behaviours and implementations of connectors, the mediator must provide a concrete solution to coordinate the interaction patterns of these connectors acting as middleware, which not only makes the application agnostic to the operating systems, but also to the middleware used by other components.

For example, Figure 2.5 depicts a mediator between *C2* and *Weather Station*. This mediator translates the $getWeather$ action required by *C2* into the $getTemperature$ and $getHumidity$ actions provided by *Weather Station*. The mediator also ensures that whenever *C2* requires $getWeather$, both $getTemperature$ and $getHumidity$ are executed. Finally, the mediator transforms the SOAP requests emitted by *C2*

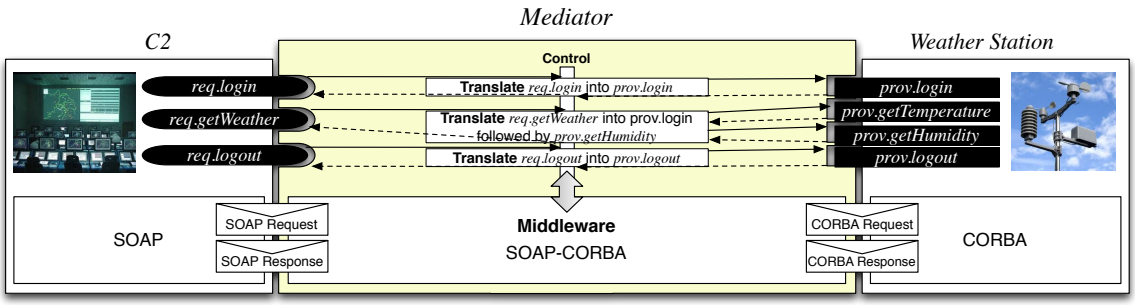

Figure 2.5. Mediator solving an architectural mismatch between *C2* and *Weather Station*

into CORBA requests that can be processed by *Weather Station*, and transforms the CORBA responses sent by *Weather Station* into SOAP responses expected by *C2*.

It is not always possible to find an existing connector for managing interaction about functionally-compatible components and it is difficult and time consuming to design and implement a new connector from scratch, especially if the components already exist and are implemented using different middleware solutions [MDT03]. Compositional approaches for connector construction facilitate the development of mediators by reusing existing connector instances.

Spitznagel and Garlan [SG03] introduce a set of transformation patterns (e.g., data translation and action aggregation), which a developer can apply to basic connectors (e.g., RPC or data stream) in order to construct more complex connectors. The authors use the approach to enhance the reliability of component interactions, but state that this approach can also be used to construct mediators that solve architectural mismatches. Each transformation pattern is given a formal definition in FSP, which allows the verification of the properties of the resulting connectors. As developers are responsible for defining the transformation patterns, they must specify both the necessary translations and behavioural coordination that must be performed by the mediator, but they can easily verify that the mediator produced ensures that the interaction between components is free from deadlocks. The approach is also equipped with a tool that facilitates the implementation of mediators by reusing and composing the implementation of existing connectors, assuming existing connectors were implemented using the same middleware.

Inverardi and Tivoli [IT13] define an approach to compute a mediator that composes a set of pre-defined patterns in order to guarantee that the interaction of components is deadlock-free. These patterns represent simple mechanisms that the mediator executes to solve differences between the interfaces or behaviours of components and consist of: (i) renaming an action, (ii) translating one action into a sequence of actions, (iii) translating a sequence of actions into one action, (iv) re-ordering sequences of actions, (v) dropping an action, and (vi) introducing a new action. This last pattern has to be taken with reserve as it implies that the mediator is able to produce an action. The mediator either only replays the action or it can perform extra computation; the latter case being beyond interoperability achievement. However, the specification of the patterns to be used must still be done by the developers. Indeed, developers specify the necessary translations based on which the approach synthesises

the mediator that coordinates the behaviours of the components. Furthermore, the implementation of the resulting mediator is completely left up to the developer as the mediator is generated from scratch without reusing existing connector implementations. The approach also ignores differences at the middleware layer.

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

2.1.3 Mediation in Ubiquitous Computing Environments

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

Chang *et al.* [CMP09] define a framework that allows component developers to define connectors, called *healing connectors*, to recover from common failures of the component. The healing connectors enable the component to operate in environments that do not verify the assumptions made during the design and implementation of this component. At runtime, whenever an exception rises due to the misuse of a component, the framework deploys, on the fly, the corresponding healing connector. The framework also maintains a log of the exceptions in order to help developers create new healing connectors. Denaro *et al.* [DPT09] apply the same approach to detect and repair disparities in different implementations of standard Web 2.0 APIs. The healing connectors are not defined by the developers but are included in a centralised catalogue that inventories the common errors that may occur when the API is used.

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

translator should be modelled explicitly in order to allow computers to reason about the information exchanged by the components and infer the translations automatically.

Another issue relates to the evaluation of the functional compatibility between components; it is unreasonable to make (or try to make) any two components work together. Interoperability should only be considered if one component requires a functionality that the other component provides. When interoperability is achieved automatically, this intuitive notion must be made precise. The explicit modelling of domain knowledge and the definition of component functionality is often supported by ontologies, as explained in Section 2.4.

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

2.2 The Middleware Perspective: Implementing Mediators

Middleware provides an abstraction that facilitates communication and coordination between components in distributed systems. It naturally follows that middleware plays a crucial role in the implementation of connectors [MDT03]. Mediators being connectors, their implementation should also be realised using middleware.

Traditionally, middleware promotes the use of a single technology based on which all components are built. However, given the diversity of modern software systems that need to be dealt with, ranging from small-scale sensors to large-scale Internet applications, there is no one-size-fits-all middleware capable of coping with them all [BPGG11]. As a result, new middleware solutions have been proposed to enable interaction across middleware and hence facilitate the implementation of mediators between independently-developed components that feature differences at both the application and middleware layers. We first present solutions based on the definition of middleware that provides developers with an abstraction which allows them to build components that are able to interact using different middleware solutions, i.e., *universal middleware*. We then consider solutions to directly translate messages from one middleware to the other, i.e., *middleware bridges*. Finally, we consider solutions to translate between different middleware solutions using an intermediary model or infrastructure, i.e., *service buses*.

2.2.1 Universal Middleware

Universal middleware solutions provide the developer with an abstraction that masks the differences that may exist at the middleware layer. For example, if *C2* was developed using some universal middleware, then it would have selected the appropriate middleware, CORBA, to communicate with *Weather Station* as depicted in Figure 2.6. We now discuss two key examples of universal middleware.

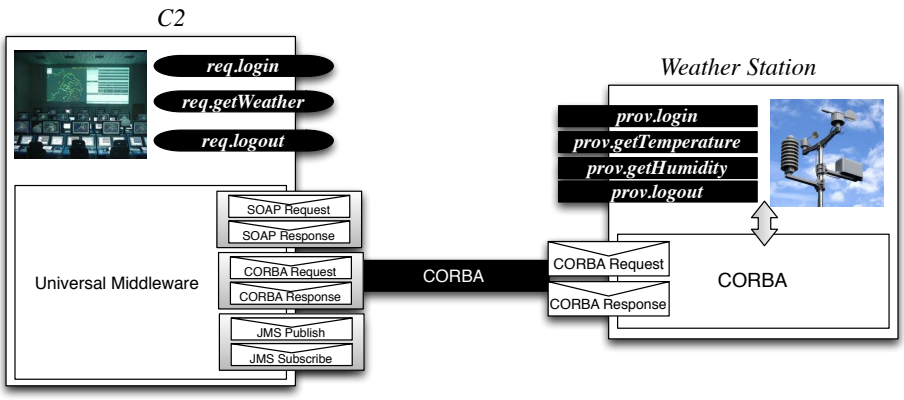

Figure 2.6. Illustrating the use of universal middleware in the weather example

PolyORB. PolyORB [VHPK04] is a middleware solution that decouples the interaction pattern used to implement the application from the middleware used for the actual achievement of this interaction. First, PolyORB supports several interaction patterns, called *application personalities*, based on which applications can be developed. Second, PolyORB supports different communication protocols called *protocol personalities*, e.g., SOAP and GIOP. The relation between the application and protocol personalities is handled via an intermediary protocol into which any application personality can be translated and which can be translated into all protocol personalities. Before deploying the component, it is configured with the appropriate personalities. Hence, it is not possible to select the appropriate protocol personality dynamically according to the running environment.

ReMMoC. ReMMoC (Reflective Middleware for Mobile Computing) [GBS03] is a reflective middleware solution that provides a WSDL-based interface to develop components. ReMMoC implements a set of plugins to transform the primitives of the WSDL interface into calls to other middleware technologies, in particular SOAP, CORBA, and STEAM (a publish/subscribe middleware). At runtime, a component implemented using ReMMoC can discover and interact with components implemented using different middleware solutions by dynamically loading the necessary plugin.

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

2.2.2 Middleware Bridges

To deal with interoperability between existing components, the most straightforward solution is to develop a middleware solution that implements direct translation between the messages of two middleware solutions. The middleware bridge takes messages from one middleware in a specific format and then marshals them to the format of the other middleware. Figure 2.7 depicts the example of interaction between *C2* and *Weather Station* using a SOAP2CORBA bridge.

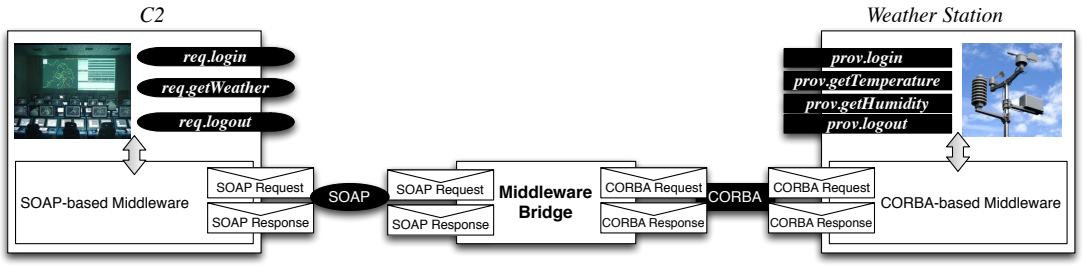

Figure 2.7. Illustrating the use of a middleware bridge in the weather example

Compiled Middleware Bridges. There exist several examples of middleware bridges: OrbixCOMet[1] is a middleware bridge between DCOM and CORBA and SOAP2CORBA[2] ensures interoperability between SOAP and CORBA in both directions. However, the implementation of middleware bridges is a complex task: developers have to deal with a lot of details involving the format of the messages used by each middleware and their correlation; therefore, developers must have a thorough understanding of the middleware at hand. As a result, solutions that help developers define middleware bridges have emerged. These solutions consist in defining a framework whereby the developer provides a declarative specification of the message translation between middleware, based on which the actual transformations are computed. z2z [BRLM09] introduces a domain-specific language to describe the message format and the communication protocol of each middleware as well as the translation logic to make them work together, and then generates the corresponding bridge. The approach has several benefits. First, it increases the level of reusability as the developer can use the individual specifications of middleware to develop different bridges. Second, the developer does not have to deal with all the message fields since z2z is able to complete default and optional fields automatically. Finally, z2z verifies that all the required fields of a message have been treated before sending it. However, the bridge cannot be modified at runtime.

Interpreted Middleware Bridges. Starlink [BGR11] uses the domain-specific models defined by z2z to specify bridges, but it deploys and interprets them at runtime. More specifically, Starlink uses the message specification associated with each

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

[2] http://soap2corba.sourceforge.net/

middleware to generate a *parser*, which is able to process the messages sent using this middleware into an *abstract message*, and a *composer*, which is able to produce the appropriate middleware message out of an abstract message. In other words, parsers and composers mask the differences between middleware through the concept of abstract messages. The translation logic specifies how to convert the abstract messages of one middleware into abstract messages of the other middleware. This approach decouples the detailed specification of the middleware, which is used to generate the corresponding parsers and composers, from the abstract specification of the translations between middleware solutions.

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

2.2.3 Service Buses

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

Middleware Buses. Georgantas *et al.* [GIBM+10] define an approach where the developer specifies a set of semantic events common to different middleware. Then, each middleware is associated with a parser that processes the messages of this middleware to produce a semantic event, and a composer that generates a middleware message based on a semantic event. Parsers and composers of different middleware then synchronise based on shared semantic events. For example, to achieve interoperability between SOAP and CORBA, developers define the request and response events. Then, parsers and composers are created per protocol: a SOAP parser triggers a request (respectively response) event upon the reception of a SOAP request

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

Enterprise Service Buses. Enterprise Service Buses (ESBs) represent the most mature and widespread solution to enable components using different middleware to interoperate, as is shown by the large number of available industrial implementations, e.g., Oracle Service Bus[1] and IBM WebSphere Enterprise Service Bus[2]. An ESB [Men07] is an open standard, message-based middleware solution that facilitates the interactions of disparate distributed applications and services. ESBs generally include built-in conversion across standard middleware technologies (e.g., SOAP, JMS) and provide a set of predefined patterns that can be used to create customised mediators. Figure 2.8 illustrates the use of an ESB to achieve interoperability between *C2* and *Weather Station*. The integration patterns facilitate the development of mediators while the SOAP and CORBA plugins help to implement these mediators between components using different middleware.

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

[1] http://www.oracle.com/technetwork/middleware/service-bus/
[2] http://www-01.ibm.com/software/integration/wsesb/

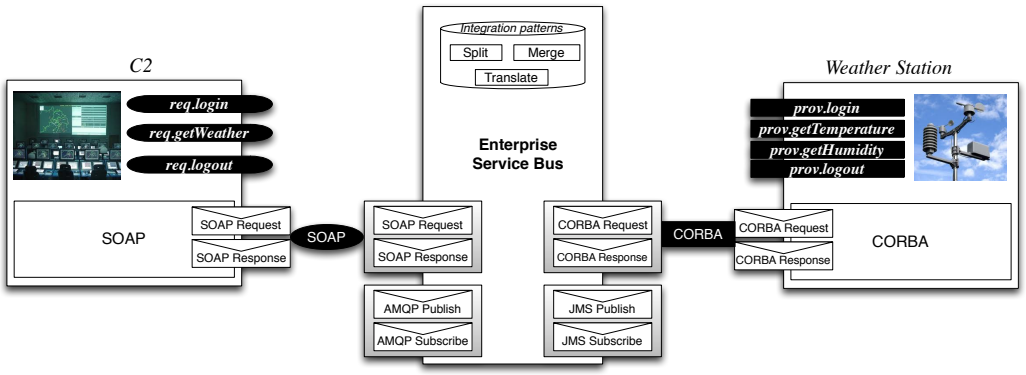

Figure 2.8. Illustrating the use of an ESB in the weather example

Analysis. There exist many middleware solutions to achieve interoperability between functionally-compatible components that feature differences at the middleware layer. However, while the implementation of new middleware might be sufficient to deal with the differences at the middleware layer, it is insufficient for dealing with differences at the application layer. First, even applications developed using the same middleware are not guaranteed to work together so long as there are differences in their interfaces and behaviours. This is, for example, the case of interoperability in Web Services [NBCT06]. Even though both services and clients use SOAP middleware, the differences between their interfaces, which include differences in the operation names, input/output message names and types, the granularity of operations, and the order in which these operations are invoked (or expected to be invoked) hamper independently-developed clients and Web Services from working together. Given the countless number of potential cases where a mediator is necessary, any static solution is doomed to fail. We need to generate mediators automatically. Second, while in the case of middleware obeying the same interaction pattern, it suffices to translate the messages sent using one middleware into messages expected by the other middleware, when middleware solutions follow different interaction patterns, e.g., a shared memory and publish/subscribe, the differences can only be solved by considering the characteristics of the applications [CMP08]. Hence, it is necessary to define solutions that are able to reason about the characteristics of applications automatically in order to synthesise the mediator that reconciles the differences between component implementations and enables them to interoperate. In the following section, we present solutions to analyse the behaviours of the components and semi-automatically generate the appropriate mediator that enables their correct interaction.

2.3 The Formal Methods Perspective: Synthesising Mediators

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

2.3.1 Mediator Synthesis Using a Specification of the Composed System

The successful interaction of components results in a composed software system that has specific properties or achieves certain user requirements. By enabling functionally-compatible components to interact with each other, even though they were not designed and implemented to do so, mediators can be seen as the missing behaviour necessary to achieve the specific properties or user requirements of the composed system.

Going back to the weather example, a user may specify that in the software system composed of *C2* and *Weather Station*, whenever *C2* invokes $getWeather$, *Weather Station* executes both $getTemperature$ and $getHumidity$, which results in the definition of a *Goal* property as follows:

$$
\begin{array}{lcl}
Req & = & (C2.req.getWeather \rightarrow P1), \\
P1 & = & (WeatherStation.prov.getTemperature \rightarrow WeatherStation.prov.getHumidity \rightarrow Req \\
 & | & WeatherStation.prov.getHumidity \rightarrow WeatherStation.prov.getTemperature \rightarrow Req).
\end{array}
$$

$$
\textsf{property} \quad \|Goal = Req.
$$

Figure 2.9 illustrates the approach on the weather example: $C2\_weather\_port$ defines the observable behaviour of *C2*, $WeatherStation\_port$ defines the observable behaviour of *Weather Station*, and *Goal* represents a user-defined specification of the composed system. The aim is to compute the process M such that the composition

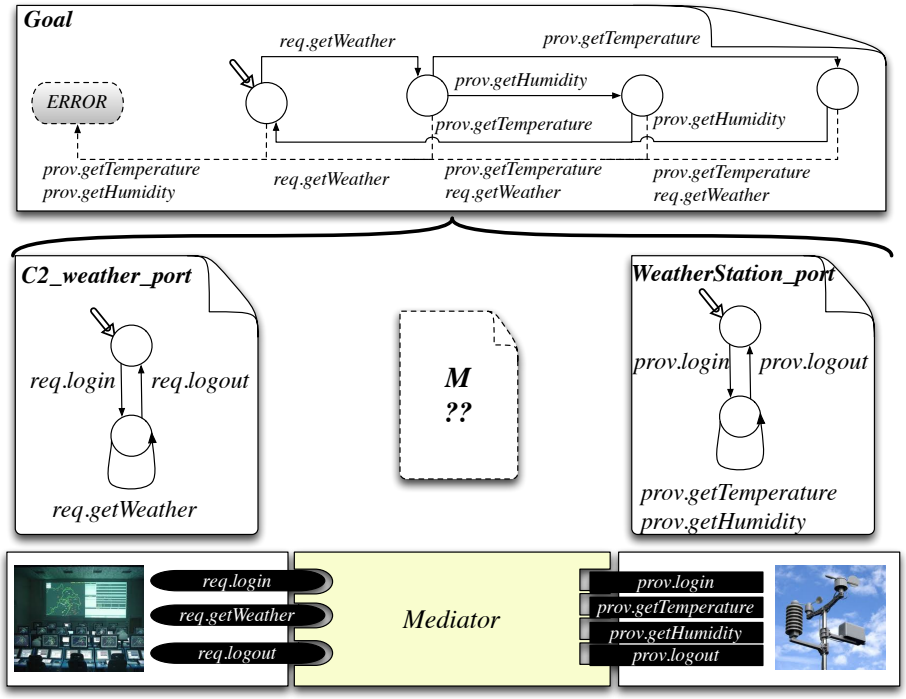

Figure 2.9. Illustrating the synthesis of mediators using a specification of the composed system

$(C2\_weather\_port \,\|M\|\, WeatherStation\_port)$ satisfies *Goal*. In this case, M represents the behaviour of the mediator. In the following, we describe approaches to synthesise the mediator M based on the definition of the behaviours of two functionally-compatible components P_1 and P_2 together with a specification of the composed system *Goal*.

Quotient. Calvert and Lam [CL89] formulate mediator synthesis as the problem of finding *quotient*. In a similar way to division and product in arithmetics, quotient can be regarded as the adjoint (roughly "inverse") of parallel composition. Given a specification for a system S, together with a component's behaviour P, the quotient yields the behaviour Q such that $P\|Q$ satisfies S. Applied to mediator synthesis, the mediator M is the quotient of the specification of the composed system *Goal* and the parallel composition of the components' behaviours $P_1\|P_2$. The authors assume *Goal* to be deterministic and synthesise M by first building the set of all possible coordinations of the actions of the components' interfaces, and then keeping only

those that satisfy *Goal*. In more detail, the algorithm is as follows.

1. Define the alphabet (interface) of the mediator as the set of actions that belong to the specification of the system but not to the alphabet of $P_1 \| P_2$, i.e., $\alpha M = \alpha Goal - (\alpha P_1 \cup \alpha P_2)$. The rationale of this definition is that the interaction between the mediator and the components is hidden in the behaviour of the composed system.

2. Calculate the set of all possible traces (a.k.a., executions) over the actions of M, i.e., $\mathcal{A}_M = (\alpha M)^*$ as well as the set of all possible traces over the actions of *Goal*, i.e., $\mathcal{A}_{Goal} = (\alpha Goal)^*$.

3. Build the trace set that represents the behaviour of M as follows: let t be a trace in $\mathcal{A}_M$ and t' a trace in $\mathcal{A}_{Goal}$. if: (i) t is equal to the projection of t' onto the alphabet of M, i.e., the trace obtained by keeping in t' only actions that belong to αM is equal to t, (ii) the projection of t' onto $(\alpha P_1 \cup \alpha P_2)$ is a possible trace of P, and (iii) t' satisfies *Goal*, then t belongs to the trace set of M.

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

Planning. Similarly to quotient computation, the planning-based approach defined by Bertoli *et al.* [BPT10] builds the mediator by identifying among all possible interactions with the components, only those that satisfy *Goal*. Nevertheless, they optimise the search by using a heuristic in order to explore only the interactions that are likely to satisfy *Goal* and use a planning algorithm in order to calculate the traces of M more efficiently.

Controller Synthesis. Gierds *et al.* [GMW12] formulate mediator synthesis in terms of controller synthesis. Besides the components' behaviours and the specification of the composed system, they also require the definition of a set of translation patterns between the actions of the components. They create a component whose

behaviour E is extracted from the specified translation patterns: E represents the behaviour of a component able to execute the translation patterns in any order. Then, they use available tools for controller synthesis to generate a controller C for the composition $P_1 \| P_2 \| E$ to satisfy *Goal*. Finally, they compose the behaviour of the controller together with the behaviour of the translation component to obtain the mediator, i.e., $M = E \| C$.

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

2.3.2 Mediator Synthesis Using a Partial Specification

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

Going back to the weather example, we can specify that the $getWeather$ action required by *C2* corresponds to the sequence of the $getTemperature$ and $getHumidity$ actions provided by *Weather Station*. Based on this specification together with the behaviours of *C2* and *Weather Station*, a mediator can be synthesised, as depicted in Figure 2.10.

Projection. Lam [Lam88] defines an approach for the synthesis of mediators based on the technique of *projections*. A projection of a component's behaviour P, noted

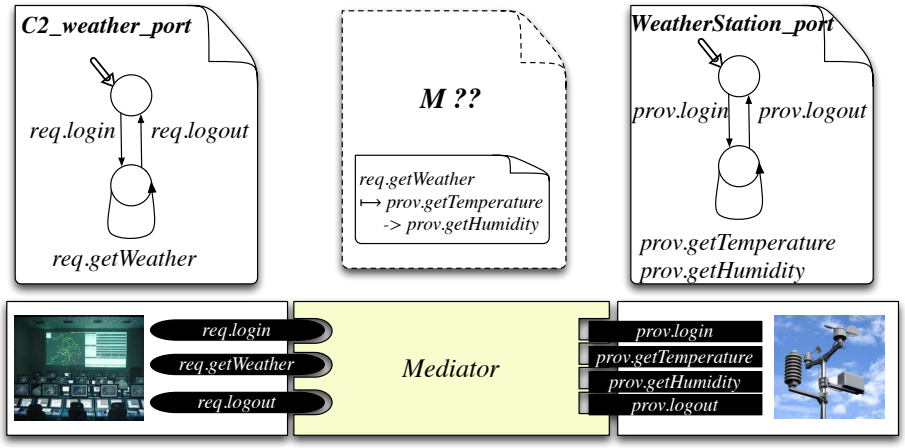

Figure 2.10. Illustrating the synthesis of mediators using a partial specification of the mediator

$Proj[P]$ is performed by aggregating some of its states, which induces the definition of an equivalence relation on the actions of the component's interface. Two actions are equivalent if they cause identical state change in $Proj[P]$ while actions that do not cause any state change are not represented in $Proj[P]$. Hence, the projection can be seen as applying relabelling and hiding functions to P.

If one can define a *useful* common projection of the behaviours of the two components P_1 and P_2, then a stateless mediator M can be synthesised. Useful means that the common projection defines a behaviour to achieve some functionality of interest. Both the definition of the common projection and the functionality are the responsibility of the developer. The stateless mediator simply transforms an action required by one component into an action provided by the other component if they cause identical state change in the common projection, and ignores the actions that do not cause any state change. However, the stateless mediator is able to deal with only one-to-one correspondences between actions. It cannot, for example, manage interaction between *C2* and *Weather Station*. Furthermore, no systematic approach for the definition of the common projection is proposed, it solely depends on developers and their understanding of the components' behaviours.

Interface Mapping. Yellin and Strom [YS97] define a synthesis algorithm that, besides the behaviours of the components P_1 and P_2, must be given an interface mapping S, which specifies the correspondence between the actions of the components'

interfaces. The interface mapping is required to be complete and non-ambiguous. An interface mapping is complete if for every required action of one component, there corresponds a provided action from the other component. It is non-ambiguous if for every required action of one component, there corresponds at most one provided action from the other component. Each correspondence in the interface mapping defines an ordering constraint between the required and provided actions. The synthesis algorithm constructs a mediator in two main phases. During the first phase, an initial process A is created which represents all possible coordinations of P_1 and P_2 that verify the ordering constraints imposed by the interface mapping. In the second phase, any execution in A leading to a deadlock is removed. As a result of the second phase, either A is empty, in which case the mediator does not exist, or it is a valid mediator M.

Model Checking. While interface mapping only specifies one-to-one correspondences between actions, there often exist more elaborate correspondences relating them. In the general case, a sequence of actions of one component may be translated into another sequence of actions of the other component. To specify complex correspondences, Mateescu et al. [MPS12] use an *adaptation contract*, which is an LTS S whose alphabet is a vector composed of the actions of the components' interfaces. The authors then construct the mediator by selecting among all possible executions of the composed system $C = P_1 \| S \| P_2$ only those that do not lead to deadlocks. Instead of constructing C then removing the erroneous executions, they use on-the-fly model checking to prune, as early as possible, the executions leading to deadlocks.

Semi-automated Mapping Generation. Nezhad *et al.* [NBM+07, NXB10] define a semi-automated approach to the synthesis of mediators which, rather than considering that the correspondences between the actions of the components are provided, define a series of heuristics to facilitate their computation. First, they focus on the syntax, expressed using XML schema, of the data embedded in the actions. They use existing XML schema matching techniques to evaluate the degree of similarity between sequences of actions in the components' interfaces. Then, they update this similarity based on the first position at which the actions can appear in the behaviours of the components: the similarity score of required and provided actions increases if they are at the same position. The last heuristic consists in selecting the

pair of actions with the highest degree of similarity according to the matching of their XML schema and then updating the similarity scores of the other pairs of actions according to their positions relative to the selected pair of actions. The same pair of actions is never selected twice so that the heuristic is guaranteed to terminate. Once the correspondences between actions have been computed, the behaviours of the two components are simultaneously explored in order to identify possible deadlocks. The user is presented with the deadlocks that may occur and has to figure out the appropriate translations that may solve them. The algorithm cannot apply the mapping directly as there is no guarantee that even the actions with the highest similarity score have the same meaning.

Analysis. Formal methods enable a rigorous analysis of components' behaviours in order to synthesis the mediator that allows the components to interoperate. Nevertheless, besides the description of components' behaviours, the synthesis of mediators using formal methods also requires the specification of the properties of the composed systems or the correspondence between actions. The properties of the composed system are hard to define by regular users, who only seek to make use of the services surrounding them. The definition of the correspondences between the actions of components' interfaces may also be error-prone, and perhaps as difficult as providing the mediator itself, given the size and the number of parameters of the interfaces involved. For example, the Amazon Web Service[1] includes 23 operations and no less than 72 data type definitions and eBay[2] contains more than 156 operations. Given all possible combinations, methods that automatically compute these correspondences are necessary.

2.4 The Semantic Web Perspective: Mediation at Runtime

A crucial part in the synthesis of mediators is defining the correspondences between the actions of the components' interfaces. When the components are dynamically discovered, and interact spontaneously, as is the case in ubiquitous computing envi-

[1] http://soap.amazon.com/schemas2/AmazonWebServices.wsdl
[2] http://developer.ebay.com/webservices/latest/ebaysvc.wsdl

ronments, the synthesis of mediators must take place at runtime. In this case, the correspondences between the actions of components' interfaces must also be elicited at runtime since the components to be mediated are not known beforehand. To do so, the meaning of these actions and their relations must be made explicit in order to allow their automated analysis. Furthermore, since users should not be involved, a systematic approach to identify functionally-compatible components must be defined. Similarly to the meaning of actions, the meaning of the functionality of a component must also be made explicit.

In this direction, the Semantic Web [BLHL+01] promotes the view that Web resources are augmented with machine-processable metadata expressing their meaning. This vision is supported by ontologies, which provide a machine-processable means to represent and automatically reason about the meaning of data based on the shared understanding of the domain [Gru09]. After introducing ontologies, we describe how, by relying on ontologies, Semantic Web Services improve the discovery, composition, and mediation of Web Services. Finally, we present a solution that promotes the convergence between ontologies and ESB.

2.4.1 Ontological Modelling and Reasoning

Ontologies allow experts to formalise knowledge about domains as a set of axioms that make explicit the intended meaning of a vocabulary [Gua04]. Hence, besides general purpose ontologies, such as dictionaries (e.g., WordNet[1]) and translators (e.g., BOW[2]), there is an increasing number of ontologies available for various domains such as biology [ABL+07], geoscience [RP05], and social networks [GR08]. The increasing number of available ontologies has further fostered the development of search engines for finding ontologies on the Web [dN12].

Ontologies are supported by a logic theory to reason about the properties and relations holding between the various domain entities. In particular, OWL[3] (Web Ontology Language), which is the W3C standard language to model ontologies, is based on description logics. More specifically, we focus on OWL DL, which is based on a specific description logic, $\mathcal{SHOIN}(\mathcal{D})$ [BCM+03]. In the rest of this document, DL refers to this specific description logic. Although traditional formal specification

[1] http://www.w3.org/TR/wordnet-rdf/
[2] http://BOW.sinica.edu.tw/
[3] http://www.w3.org/TR/owl2-overview/

techniques (e.g., first-order logic) might be more powerful, DL offers crucial advantages: it excels at modelling domain-specific knowledge while providing decidable and efficient reasoning algorithms.

DL specifies the vocabulary of a domain using concepts, attributes of each concept, and relationships between these concepts. Each concept is given a definition as a set of logical axioms, which can either be atomic or defined using different operators such as disjunction ($C \sqcup D$), conjunction ($C \sqcap D$), and quantifiers ($\forall R.C$, $\exists R.C$) where C and D are concepts and R is an object property .The attributes of a concept are defined using an object property, which associates the concept with a built-in data type. DL can also be used to describe the aggregation of concepts using the W3C recommendation for part-whole relations[1] (hasPart), where different concepts are composed together to build a whole. A concept E is an aggregation of concepts C and D, written $E = C \oplus D$, provided both C and D are parts of E, i.e., $E = \exists \mathsf{hasPart}.C \sqcap \exists \mathsf{hasPart}.D$. For example, the Weather concept is defined as the aggregation of the Temperature and Humidity concepts, meaning that each weather instance $t \in \mathsf{Weather}$ encompasses a temperature instance ($\exists f \in \mathsf{Temperature} \wedge (t, w) \in \mathsf{hasPart}$) as well as a humidity instance ($\exists h \in \mathsf{Humidity} \wedge (w, h) \in \mathsf{hasPart}$). The syntax and semantics of DL operators are summarised in Appendix B, while the interested reader is referred to [BCM+03] for further details.

Traditionally, the basic reasoning mechanism in DL is *subsumption*, which can be used to implement other inferences (e.g., satisfiability and equivalence) using predefined reductions [BCM+03]. Intuitively, if a concept C is subsumed by a concept D (written $C \sqsubseteq D$), then any instance (also called individual) of C also belongs to D. In addition, all the relationships in which D instances can be involved are applicable to C instances, i.e., all properties of D are also properties of C. Subsumption is a partial order relation, i.e., it is reflexive, antisymmetric, and transitive. The result is that the ontology can be represented as a hierarchy of concepts. It is important to note that the hierarchy is not created manually. Rather, each concept is given a definition as a set of logical axioms. An ontology reasoner infers a hierarchy of concepts. This offers more flexibility by making the hierarchy evolve naturally as new concepts are added. Furthermore, there exist efficient reasoners to automate the inference task [DCtTdK11].

[1] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/

2.4.2 Semantic Web Services

Web Services are processes that expose their interfaces to the Web so that users can invoke them. Semantic Web Services provide a richer and more precise way to describe the services through the use of knowledge representation languages and ontologies. In the following, we present the two major efforts for modelling and using Semantic Web Services.

OWL-S (Semantic Markup for Web Services). OWL-S [MBM+07], which was previously named DAML-S [BHL+02], is an ontology for formally defining Web Services. As depicted in Figure 2.11, the OWL-S ontology is structured around three main concepts: Profile, Process, and Grounding. The service profile describes what the service does, i.e., the functionality it provides to its clients. The service profile, also called service *capability*, has a name and a textual description, and is associated with a category, which refers to a concept in an ontology of service categories. The profile is also associated with some inputs (Input), outputs (Output), pre-conditions (Condition), and post-conditions (Result). Inputs and outputs are defined as the set of all the inputs and outputs with which the service process is associated while pre- and post-conditions are selected among the pre- and post-conditions of the service process. The service *process* describes the service's behaviour, i.e., how the service achieves its capability. A process has some input, which defines the information necessary

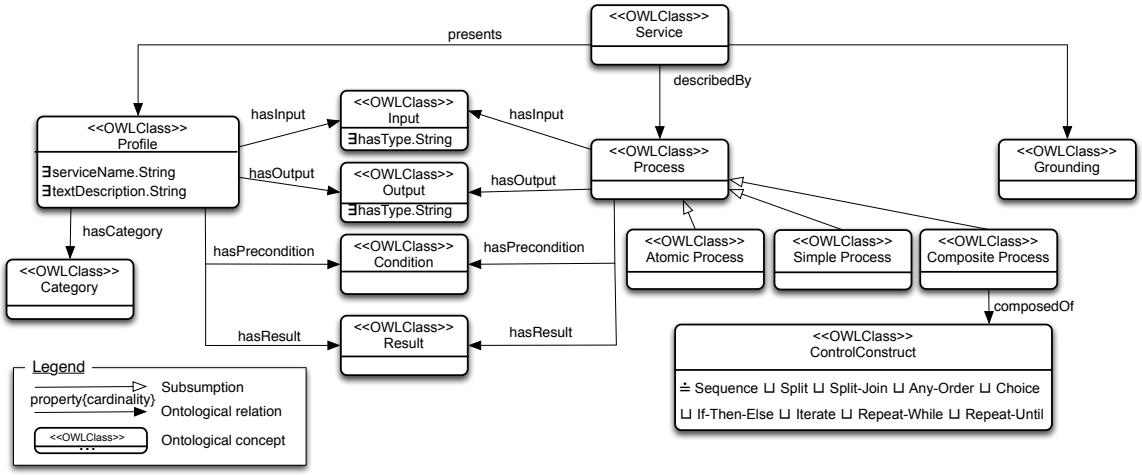

Figure 2.11. Extract of the OWL-S Ontology

for its execution, and produces some output. The Input and Output concepts are associated with a URI that defines the type of the input/output data. A process is also associated with some pre- and post-conditions. A process can be Atomic, if it can be invoked, Simple if it does not have a corresponding grounding, or Composite, if it is composed of other processes using some ControlConstructs. The control constructs are similar to those used in process algebra for the specification of the component's behaviour, with the exception of the parallel composition that is not supported as a control construct. The service grounding describes the information necessary to invoke the service.

An ontology-based description of Web Services has many advantages. First, it promotes the discovery of functionally-compatible components through the notion of a capability. In this sense, pioneering work by Paolucci *et al.* [PKPS02] defines an approach to assess functional compatibility between a provided service (advertisement) and a required service (request) by comparing the semantics of the inputs and outputs specified in their respective profiles: an advertisement matches with a request if every input in the request profile subsumes some input in the advertisement profile, and every output in the advertisement profile subsumes some output in the request profile. Second, it eases the construction of composition of services by making explicit the input, output, pre- and post-conditions of the services as well as their behaviours. Finally, and most importantly, it facilitates mediation by formalising both the meaning of the input/output and the behaviour of services. Vaculín *et al.* [VNS09] define an approach for generating mediators between functionally-compatible client and service, both of which are modelled using OWL-S. First, they extract a set of representative executions of the client using its process specification. For each execution, they simulate the service process and use a planning algorithm in order to find the corresponding execution such that the client and the service can progress simultaneously. Then, for each pair of client and service executions, they use existing data mediators to perform the translations necessary to compensate for the differences between their input/output.

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

WSMO (Web Service Modelling Ontology). WSMO [CM05] is an ontology for modelling Semantic Web Services. As illustrated in Figure 2.12, it is based on four main concepts: Web Service, Goal, Ontology and Mediator. *Web Service* defines the computation entity providing some functionality, which is described using the Capability concept. The semantics of the capability is given in terms of several axioms: pre- and post-conditions, assumptions, and effects. Goal describes the computation entity requiring some functionality. Interface defines the behaviour of the Web Service or the Goal with Choreography describing the necessary information to interact with the service from the client point of view, and Orchestration describing how the service makes use of other services in order to achieve its capability. The Ontology concept represents the semantics of the concepts used for the definition of the WSMO element, which can be a Web service, goal or mediator. The Mediator concept specifies how differences between WSMO elements are solved: ooMediator for differences between ontologies, wwMediator for differences between services, wgMediator for differences between a service and a client, ggMediator for differences between clients. The implementation of mediators is supported by a runtime framework, the Web Service Execution Environment (WSMX), which executes the specified mediators, thereby allowing independently-developed clients and services to interoperate.

As for OWL-S, the formal description of the capabilities and behaviours of services facilitates the discovery and composition of services. But WSMO has also several differences with OWL-S. First, there is a clear distinction between service provision

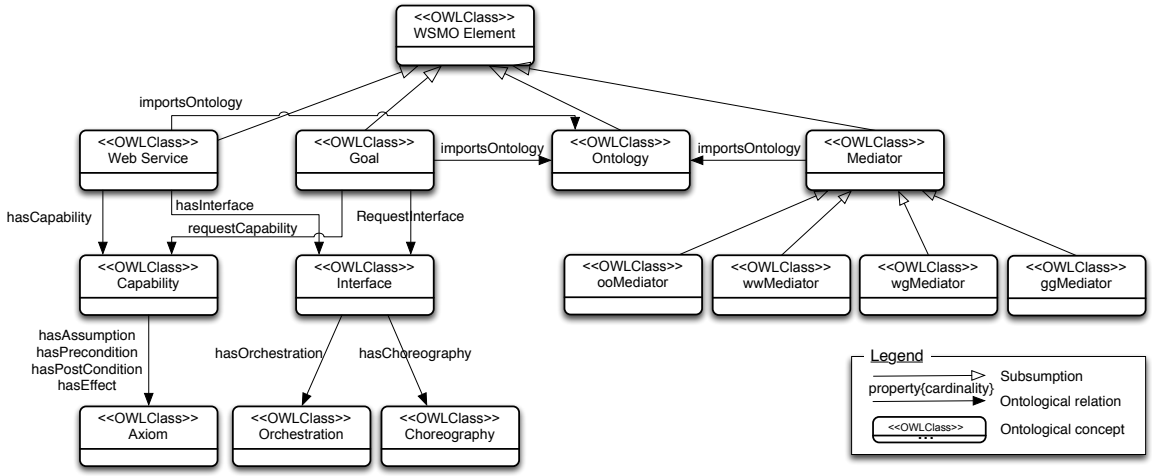

Figure 2.12. Extract of the WSMO Ontology

represented using the Web Service concept, and service consumption represented using the Goal concept. Second, specific concepts are used to define ontologies and mediators.

2.4.3 Semantic Mediation Bus.

Instead of defining yet another ontology for Web Services, SA-WSDL [KVBF07] proposes a cost-effective solution to incorporate ontology reasoning in Web Services by augmenting service descriptions with annotations to: (i) define the semantics of operations and data by referring to concepts in a domain ontology, (ii) map the data syntax to the semantic definition of the associated concept using XSLT[1], i.e., *lifting*, and (iii) derive the specific data structures from semantic concepts using XSLT also, i.e., *lowering*.

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

The Alion Semantic Mediation Bus [Zhu12] brings together SA-WSDL and ESB. While the ESB provides various plugins to support different middleware interaction protocols, services are described using SA-WSDL specifications, which enables the runtime translation of the actions of clients' and services' interfaces using the lifting and lowering functions. Nevertheless, as SA-WSDL does not support the modelling of behaviour, the Alion Semantic Mediation Bus focuses on action translations and does not coordinate the behaviours of clients and services. Moreover, as the capabilities are not represented either, the discovery of functionally-compatible clients and services cannot be achieved automatically.

Analysis. Semantic Web technologies, and ontologies in particular, enable the precise modelling of and reasoning about the meaning of the information exchanged between components. Semantic Web Services illustrate how ontologies can help to automate the discovery and composition of Web Services and facilitate mediation between them. In addition, through the capability concept, functionally-compatible components can be selected automatically. However, mediation is often based on the definition of new ontologies and their use to infer the translations necessary to ensure the meaningful exchange of information between components. Furthermore, while

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

modelling the behaviour of components is recognised as being essential, the logical theory behind ontologies is inappropriate for analysing components' behaviours. In addition, even though initial attempts to handle differences between components at the middleware layer are beginning to emerge through the concept of semantic mediation buses, they only deal with translations of actions and do not manage behavioural differences between components, at either the application or the middleware layers.

2.5 Summary

Interoperability is a very challenging topic. Over the years, interoperability has been the subject of a great deal of work, both theoretical and practical. First, to understand and formalise interoperability, then to implement mediator-based solutions to achieve it with an increasing shift towards runtime. In this chapter, we surveyed the solutions to interoperability from its four underpinning perspectives. Table 2.1 summarises the solutions presented in this chapter. We can notice that none of the proposed solutions is able to synthesise and implement mediators that deal with both application and middleware differences and guarantee that the interaction between functionally-compatible components is error-free.

Interoperability is a complex challenge that can only be solved by appropriately combining different techniques and perspectives. These techniques include formal approaches for the synthesis of mediators with the support of ontology-based reasoning so as to automate the synthesis, together with middleware solutions to realise and execute these mediators. In the next chapter, we introduce a multifaceted approach to interoperability, which brings together and enhances the solutions that tackle interoperability from different perspectives.

Pers.	Approach	The main idea	Evaluation
Software Architecture	Formal Reasoning about interoperability [GAO95, AG97]	Formal definition of component interaction to detect architectural mismatches	+ Formal basis for understanding interoperability — No support for differences at the middleware layer
	Compositional approaches for connector development [SG03, IT13]	Creating mediators from existing connector instances	— No automated generation of mediators
	Self-healing connectors [CMP09, DPT09]	Recovery from component misuse by deploying connectors on the fly	
Middleware	Universal middleware [GBS03, VHPK04]	Provide an abstraction that masks the differences at the middleware layer	+ Support differences at the middleware layer — Developers need to specify or implement mediators at the application layer
	Middleware bridges [BRLM09, BGR11]	Direct translation between middleware messages	
	Service buses [Men07, GIBM+10]	Dealing with different middleware solutions via an intermediary infrastructure	
Formal Methods	Using a specification of the composed system [CL89, BPT10, GMW12]	Synthesise the mediator by selecting from all possible coordinations of the behaviours of components only those that satisfy the specification of the composed system	+ Automated analysis and coordination of components' behaviours + Guaranteed correctness of the interaction between components — Require a declarative specification of the correspondences between the actions of components' interfaces — No support for differences at the middleware layer
	Using a partial specification [Lam88, YS97, NXB10, MPS12]	Require the correspondences between actions to be available, and synthesise the mediator that guarantees that interaction between components is deadlock-free	
Semantic Web	Semantic Web Services [BHL+02, CM05, MBM+07]	Defining an ontology to support the inference of the necessary translations of the actions required by one component and provided by the other	+ Automated discovery of functionally-compatible components + Automated reasoning about the meaning of information
	Semantic mediation bus [Zhu12]	Using semantic technologies within an ESB to automate message translation	— Partial support for behavioural differences — Partial support for middleware differences

Table 2.1. Summary of approaches to interoperability

Chapter 3

Achieving Eternal Interoperability: The Role of Automated Mediator Synthesis

> *"The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything in any form of language."*
>
> — in H2G2 by Douglas Adams, author and satirist, (1952-2001)

Despite extensive interest and intensive work, interoperability remains an open problem, especially in ubiquitous computing environments. The previous chapter highlighted the fact that a great deal of progress has been made in achieving interoperability by providing solutions for the synthesis and implementation of mediators. However, existing solutions have not fully succeeded in coping with the increasing complexity of modern software systems, both in terms of the level of heterogeneity and the degree of dynamism. We claim that only a multifaceted approach that brings together and enhances the solutions that tackle interoperability from different perspectives is viable in the long term. In this thesis we define a solution to interoperability which, rather than defining yet another middleware or yet another ontology, exploits existing middleware together with knowledge encoded in existing domain ontologies to synthesise and implement mediators automatically. In this chapter, we present the context in which our solution takes its full significance. Our solution plays a central role in the approach to *eternal* interoperability among highly heterogeneous distributed systems, and systems of systems, developed within the CONNECT

project. Eternal interoperability emphasises the fact that the proposed approach is not restricted to today's components, i.e., components based on existing middleware and standards, but should be applicable to future and unforeseen ones as well. We present the principles and techniques for supporting eternal interoperability and show the key role of the automated synthesis and implementation of mediators therein.

3.1 The CONNECT Approach to Eternal Interoperability

The CONNECT project[1] aims at breaking down the interoperability barriers in highly dynamic and extremely heterogeneous environments such as ubiquitous computing environments or systems of systems. Ubiquitous computing promotes a view of components, distributed across many, possibly mobile, devices discovering one another dynamically and interacting on the fly. Systems of systems integrate existing components, which are autonomous and so complex that they can be considered as systems in themselves. The components of a system of systems are designed and implemented independently, and can be extremely heterogeneous with differences spanning the application and middleware layers, which makes their subsequent integration very complicated. Achieving interoperability in these environments is extremely challenging because the components already exist, can be very complex, are heterogeneous, and discover one another dynamically. To address this challenge, CONNECT advocates an approach to interoperability based on *emergent middleware*. Emergent middleware is a dynamically generated implementation of a mediator for the current operating environment and context, which makes functionally-compatible components interoperate seamlessly. Hence, an emergent middleware solution requires the ability to identify functionally-compatible components dynamically as well as to synthesise and implement mediators at runtime. The former can be achieved by explicitly modelling the functionality required or provided by components. The latter necessitates the precise description of the meaning of the actions required or provided by the components as well as the components' behaviours. In other words, an emergent middleware solution relies on the appropriate modelling of components.

However, while the appropriate modelling of the functionality and behaviour of

[1] http://www.connect-forever.eu

components has been recognised as being essential to reason about interoperability and synthesise mediators automatically, as presented in Chapter 2, such models are often unavailable in practice. Most components only display their syntactic interfaces, which describe, using some textual notation or XML, the actions through which the components interact. Therefore, it is often necessary to use the syntactic interface of the component to complete its model.

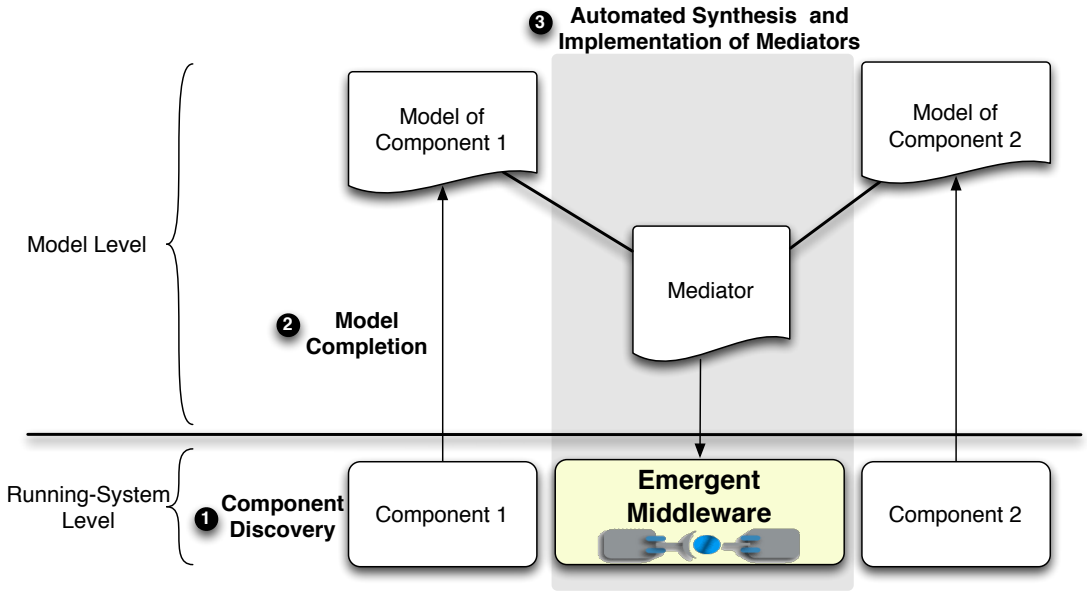

Figure 3.1. The CONNECT approach for creating emergent middleware

Figure 3.1 outlines the overall CONNECT approach for creating emergent middleware. The approach operates at two levels: the *running-system level*, where the implementations of software components are defined and the interactions between components take place, and the *model level*, where the models of software components are defined and the synthesis of mediators takes place. The first step starts at the *running-system level* by discovering the components available in the runtime environment (see Figure 3.1-❶). Most discovery protocols (e.g., SLP[1], WS-Discovery[2], UPnP-SSDP[3], and Jini[4]) only support the description of the syntactic interface of components. Therefore, the second step is to complete the component model (see

[1]Service Location Protocol – http://www.openslp.org/
[2]http://docs.oasis-open.org/ws-dd/discovery/1.1/
[3]Universal Plug and Play-Simple Service Discovery Protocol – http://www.upnp.org/
[4]http://www.jini.org/

Figure 3.1-❷). Based on the completed models of components, the third step consists in identifying pairs of functionally-compatible components and synthesising the appropriate mediator that guarantees their correct interaction. Then, the synthesised mediator is implemented as emergent middleware (see Figure 3.1-❸).

The CONNECT approach is fully automated, does not require *a priori* knowledge of the components, and creates the emergent middleware at runtime. This approach is not only suitable for achieving interoperability between today's components but also future components, which are based on yet-to-be applications and middleware solutions.

In the following sections, we introduce the component model in more detail. Then, we present the architecture of emergent middleware and describe the CONNECT architecture for generating it at runtime.

3.2 Modelling Components

We build upon the work on interoperability presented across the different perspectives, as surveyed in Chapter 2, to define a component model that captures all the information necessary to achieve interoperability automatically. We recall that both the software architecture and the formal methods perspectives require describing the behaviour of the component using a rigorous formalism. The Semantic Web perspective also promotes the precise description of the functionality and the meaning of actions in the component's interface using ontologies. Finally, the middleware must also be specified so as to facilitate the implementation of mediators. The component model is made up of *capability*, *interface*, and *behaviour*.

Capability

The component's *capability* (Cap) gives a macro-view of the component by specifying the functionality it requires from or provides to its environment. A capability is specified as: $Cap = \langle type, F \rangle$ where:

- $type \in \{Req, Prov, Req\_Prov\}$ specifies whether the component expects some functionality to be provided by the environment, i.e., Req; it produces the functionality itself, noted $Prov$; or both requires and provides the functionality, as is the case in peer-to-peer interactions.

- F gives the semantics of the functionality by referring to a concept in an ontology of domains $\mathcal{D}$, which is a general ontology representing the different categories of services components may require or provide, in a similar way to the UNSPSC taxonomy[1]. Every concept in $\mathcal{D}$ is associated with a domain ontology $\mathcal{O}$, which models knowledge about the specific domain.

A component may have several capabilities, each of which represents some functionality required or provided by the component and relates to a specific domain. For example, in the GMES case study [Con12], the Command and Control centre *C2* has to interact with different components to make the decisions necessary for crisis management; it needs to get weather information, the location of firemen, and to manage vehicles remotely. As a result, *C2* has several capabilities, each of which is related to a specific interaction.

Interface

The interface signature, or simply *interface* ($\mathcal{I}$), of the component gives a finer-grained description of the actions that the component manipulates. Actions are described both syntactically, using XML schema, and semantically using ontological annotations. Annotations offer a simple technique to include domain knowledge by simply referring to ontology concepts. The annotations of a component's interface refer to a single ontology $\mathcal{O}$ specific to the domain that the component belongs to.

More specifically, a required action $\alpha =<op, i, o>$ $(op, i, o \in \mathcal{O})$ represents an invocation of an operation op by providing the appropriate input data i and consuming the corresponding output data o. While we represent the required action using the ontology concepts representing the operation and data, the syntax of the input and output data (i and o) is given using the associated XML schema, which we denote $\mathcal{S}(i)$ and $\mathcal{S}(o)$ respectively. The dual provided action[2] $\overline{\beta} =<\overline{op}, i, o>$ uses the inputs and produces the corresponding output.

Finally, the interface also specifies the middleware used to implement the required and provided actions. When a standard middleware solution is used, e.g., SOAP, it is straightforward to realise the required and provided actions. However, when some

[1] United Nations Standard Products and Services Code – http://www.daml.org/ontologies/106

[2] We use the overline as convenient shorthand for provided actions.

proprietary solution is used or when the standard middleware is customised, for optimisation purposes for example, we rely on a domain-specific language, Message Description Language [BGR11] (MDL), to describe how the actions are realised. MDL describes the exact structure of messages to be sent on the network. Each message is described as a sequence of *fields*. The definition of each field describes its length in bits, its type, and its value if it is constant. For example, the following listing is an extract of the definition of a Yahoo! instant message:

```
<Message>
 <Name>SDG</Name>
 <Field>
  <PrimitiveField>
   <label>Tag</label>
   <length>24</length>
   <type>String</type>
   <mandatory>true</mandatory>
   <value>SDG</value>
  </PrimitiveField>
 </Field>
 <Field>
  <PrimitiveField>
   <label>User</label>
   <length>0</length>
   <type>String</type>
   <mandatory>true</mandatory>
 </PrimitiveField>
 <Field>
  <PrimitiveField>
   <label>Conversation</label>
   <length>0</length>
   <type>String</type>
   <mandatory>true</mandatory>
  </PrimitiveField>
 </Field>
 ...
 </Message>
```

It specifies that the *SDG* message includes a *Tag* field of 24 bits whose value is set to *SDG*. The two other fields, *User* and *Conversation*, are both mandatory. Their *length* is set to 0 as it is variable and described in another field.

Behaviour

The *behaviour* of a component specifies its interaction with the environment and models how the actions of its interface are coordinated to achieve its functionality. We build upon FSP and the related LTS semantics to specify the component's behaviour. The interface of the component defines the alphabet of the FSP process P representing the behaviour of the component, i.e., $\alpha P = \mathcal{I}$. We assume synchronous semantics, i.e., a component can only perform a required action a if another component is in a state that enables it to perform the dual provided action $\overline{a}$, since the required action can only be carried out if its output data are available; similarly, a component can only perform a provided action $\overline{b}$ if another component is in a state that enables it to perform the dual required action b, since the provided action can only be carried out if its input data are available. The reason is that behavioural analysis under asynchronous semantic is in general undecidable [BZ83]. Note, though, that the realisation of the actions may be asynchronous, as is the case when using a publish/subscribe middleware.

3.3 Emergent Middleware

The emergent middleware is the realisation of the mediator. In CONNECT, Starlink [BGR11] is used to enact mediators as emergent middleware. Starlink uses a *mediator engine* to interpret and execute the coordination and the translations specified by the mediator. While the mediator engine manipulates required and provided actions, these actions must be transformed into concrete network messages exchanged with the components. The format of these messages depends on the middleware used by the component. Therefore, Starlink uses a *parser* and a *composer* to communicate with each component. The parser analyses the messages received by the emergent middleware and transforms them into actions that can be understood by the mediator engine. The composer is responsible for transforming actions into messages that the emergent middleware can send. Starlink generates parsers and composers based on the MDL descriptions of how the actions are realised. While this solution is very flexible as it can deal with any middleware solution, whether it be standard or proprietary, it is somewhat tedious as it requires the developer to know and specify the exact structure of the messages sent or received by the components. Therefore, we inves-

tigated an approach whereby parsers and composers can be generated by combining and reusing existing middleware libraries. This technique is generally more efficient as it reuses optimised libraries for processing messages. Furthermore, the developer need not know the exact structure of the messages but only the middleware used.

Figure 3.2 illustrates an emergent middleware solution in the case of the weather example. The mediator engine executes the mediator in order to perform the translations and the coordinations necessary to enable *C2* and *Weather Station* to interact. To communicate with *C2*, the emergent middleware uses a SOAP parser and composer. It uses a CORBA parser and composer to communicate with Weather station.

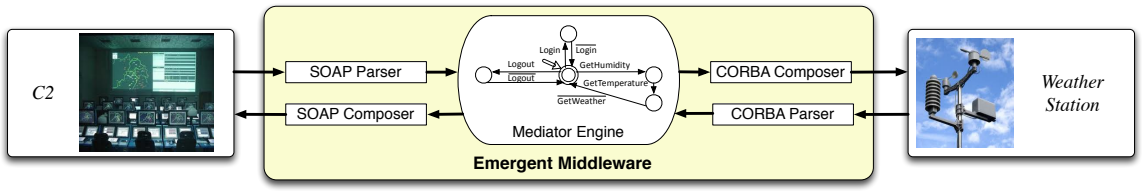

Figure 3.2. Emergent middleware between *C2* and *Weather Station*

3.4 Emergent Middleware Enablers

The CONNECT approach is implemented using several software entities, called *enablers* that collaborate in order to produce an emergent middleware solution, as depicted in Figure 3.3. The *discovery enabler* is responsible for locating the components available in the environment. When a component is discovered which only describes its syntactic interface, the discovery enabler invokes the *learning enabler* in order to complete the model of the component. Based on the completed models of components, the discovery enabler identifies functionally-compatible components and invokes the *synthesis enabler* to generate and implement the mediator that makes it possible for them to interoperate. We describe each enabler in the following.

3.4.1 Discovery Enabler: Locating Components

The discovery enabler is responsible for identifying functionally-compatible components and enabling them to discover one another, assuming that these components

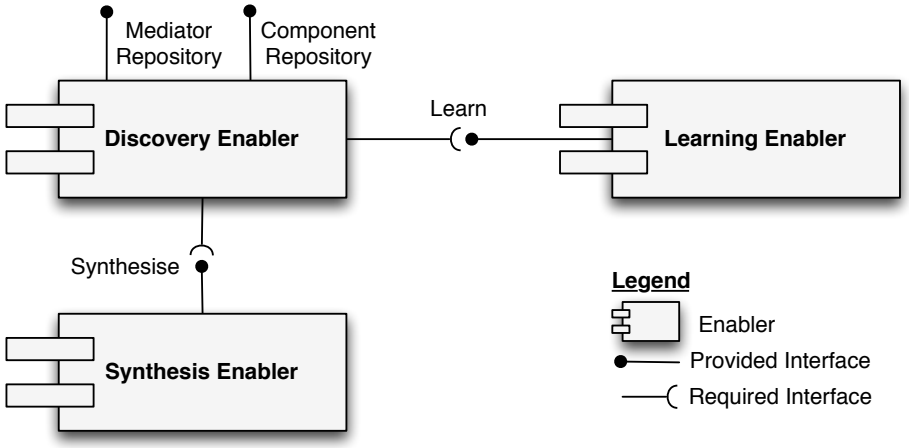

Figure 3.3. Emergent middleware enablers

make use of some discovery protocol (a.k.a. service discovery protocol) to advertise their presence in the environment and also receive the advertisements of other components.

Figure 3.4 gives an overview of the discovery enabler. First, the discovery enabler uses a set of plugins, each of which is associated with some existing discovery protocol (e.g., SLP, WS-Discovery, UPnP-SSDP, and Jini), to locate the components joining the runtime environment and which use these discovery protocols to advertise their presence. The advertisement often contains only the description of the syntactic interface of the component. Consequently, the discovery enabler uses the Learn interface provided by the learning enabler to complete the model of this component.

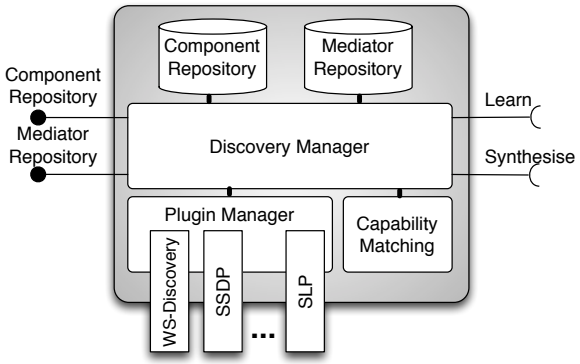

Figure 3.4. Overview of the discovery enabler

Given the completed models of components, the discovery enabler identifies functionally-compatible components using *capability matching*. More precisely, we say that a required capability $Cap_R = \langle \mathsf{Req}, Op_R \rangle$ semantically matches a provided capability $Cap_P = \langle \mathsf{Prov}, Op_P \rangle$ *iff* $Op_P \sqsubseteq Op_R$ in the ontology of domains $\mathcal{D}$. The idea behind this condition is that the functionality required by one component is less demanding that the functionality provided by the other component. Once a pair of functionally-compatible components has been identified, the discovery enabler uses the Synthesise interface provided by the synthesis enabler to generate and implement the mediator that enables them to interoperate.

The discovery enabler maintains a repository of component models and a repository mediators to which it gives access using the eponymous interfaces. Both repositories are organised according to the subsumption relation between the capabilities of components in order to accelerate the search for components' models and reduce the time necessary to identify functionally-compatible components.

3.4.2 Learning Enabler: Completing Component Models

Although capabilities and behaviours have been acknowledged as essential elements of component specification, it is the exception rather than the rule to have such rich component descriptions available on the network. Given the description of a component's interface, the learning enabler uses statistical learning to infer the ontology concept representing the functionality required or provided by the component, and automata learning to extract its behaviour.

Statistical Learning for Inferring the Component's Capability

Since the interface is typically described by textual documentation, e.g., XML documents, we capitalise on the long tradition of research in *text categorisation*. Text categorisation enables the classification of a textual document into a predefined set of categories. One of the main techniques for text categorisation is Support Vector Machines (SVM) [Joa98], which is a learning algorithm that has the ability to infer a *categorisation function* based on a set of *features*. For text categorisation, the standard representation of features is a *bag of words* [SYY74]. In this method, words are associated with dimensions of the vectors used by the SVM. For example, a textual document consisting of the string "get Weather, get Station" could be represented

as the vector $(2, 1, 1, \ldots)$ where 2 in the first dimension is the frequency of the "get" token. The SVM algorithm is first given a training set that consists of textual documents, each of which is associated with the appropriate category. As a result it produces the categorisation function, which associates a textual document to a category according to its features. We use SVM to infer the categorisation function that relates an interface, which is considered as a textual document, to a semantic concept from the ontology of domains $\mathcal{D}$, which represents the set of possible categories. At runtime, the interface is analysed in order to infer the appropriate functionality, as illustrated in Figure 3.5. Note that while learning is used to infer the ontology concept representing the functionality of the capability, the type of the capability (*Req*, *Prov*, or *Req_Prov*) is set by the discovery enabler based on whether the component is advertising its interface or looking for a component to interact with.

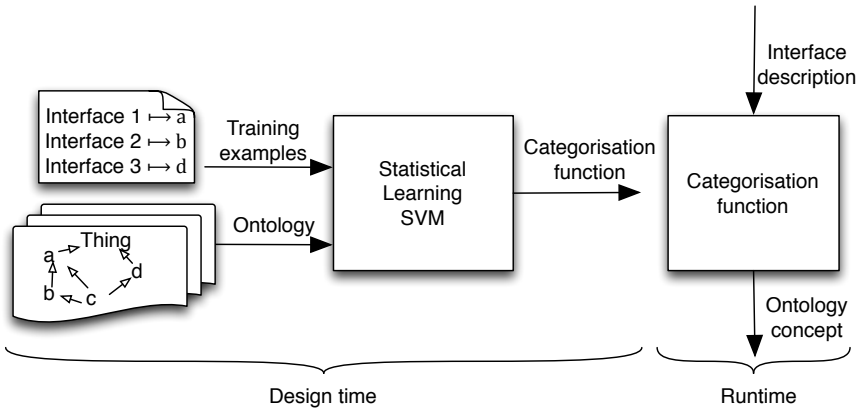

Figure 3.5. Illustrating capability learning

Automata Learning For Inferring the Component's Behaviour

A learning technique based on Angluin's seminal L^* algorithm [Ang87] is used to extract the behaviour of a component when only its interface is known. It is based on an iterative process by which a hypothesis component's behaviour is incrementally refined by actively testing interactions with the corresponding component. Hence, unlike passive learning algorithms [LMP08, KBP+10] that only observe the interaction traces, L^* chooses the sequences of actions to execute in order to learn the behaviour in minimal time. In CONNECT, the learning technique is provided by LearnLib [MSHM11], a framework for automata learning, which implements various

improvements to the L^* algorithm such as abstraction/refinement or dealing with data values in order to be able to learn the behaviour of realistic components in minimal time.

Going back to the weather example, let us consider learning *C2*'s behaviour as illustrated in Figure 3.6. At time t_0, the interface of *C2* is discovered, and the learning algorithm initiates by assuming that *C2* is able to perform any action in any order, that is its behaviour is represented using one single state where all the actions can be performed. However, when trying to interact with the system by performing, for example, *req.getWeather* and then *req.login*, an error (or exception) is raised. At time t_1, the model is updated so as to forbid this erroneous trace. Similarly, when performing *req.login*, *req.logout*, then *req.getWeather*, an error occurs, therefore the learning algorithm updates the model, which continues to be refined to obtain the model at time t_3. LearnLib verifies the data types of actions in order to refine the initial behaviour. Hence, it would directly start with the behaviour specified at t_1.

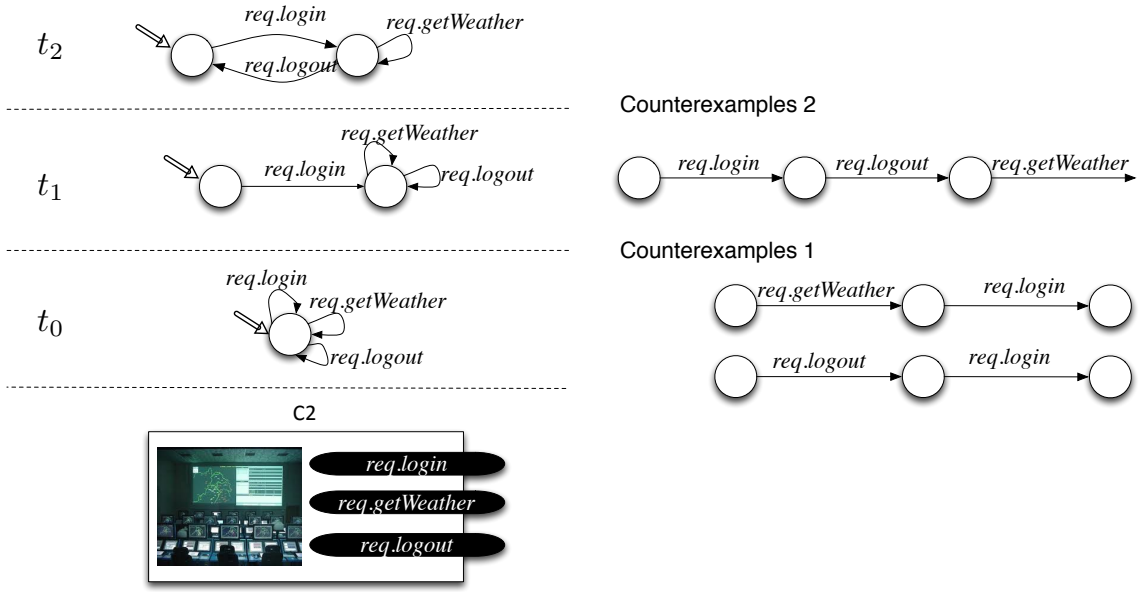

Figure 3.6. Learning the behaviour of *C2*

3.4.3 Synthesis Enabler: Synthesising Mediators

The automated synthesis of mediators between functionally compatible components, which is the main role of the synthesis enabler, lies at the heart of the creation of

the emergent middleware. It is based on our approach for the automated synthesis of mediators, which is the central focus of this thesis.

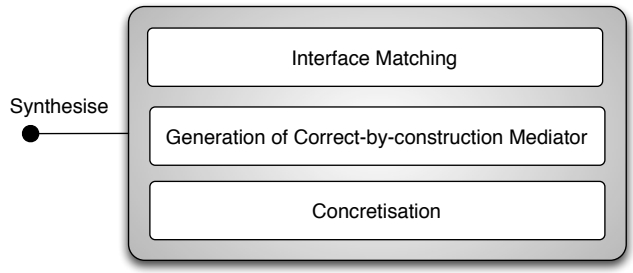

Figure 3.7. Overview of the synthesis enabler

In order to allow components to interoperate seamlessly, it is important to find the right level of abstraction so as to reason about the interaction of these systems automatically while keeping enough details to turn the conclusions drawn during the reasoning phase into a concrete artefact. It is difficult to deal with implementation-level interoperability, as it involves managing many details that, although crucial, make the reasoning very difficult, if not impossible. But an excessive abstraction is also useless as the decision space toward refining the result of the reasoning and turning it toward a concrete solution would be immense. Furthermore, knowledge about the domain in which the components evolve is necessary in order to capture the meaning of the information they exchange. Therefore, the synthesis enabler is composed of several modules (see Figure 3.7). First, we use domain knowledge, which is represented using the adequate domain ontology, to calculate the correspondences between the actions required by one component and those provided by the other, that is *interface matching*. The interface matching must be complete in the sense that every required action is involved in at least one correspondence. The interface matching might be ambiguous, that is the same sequence of required actions may correspond to different sequences of provided actions. We use the interface matching to build a *correct-by-construction mediator* that guarantees that the components are able to work together at the application layer. Finally, we *concretise* the mediator in order to take into account the characteristics of the middleware solutions used to implement the components. We detail interface matching and the generation of correct-by-construction mediators in Chapter 4 and the concretisation in Chapter 5.

3.5 Summary

In this chapter, we presented the CONNECT approach to achieve interoperability in highly dynamic environments. The approach does not require any human intervention as it discovers the components dynamically, learns their behaviours automatically, and produces the emergent middleware that enables functionally-compatible components to interoperate seamlessly. We also highlighted the role of the automated synthesis and implementation of mediators in the creation of emergent middleware. In this context, we can reasonably assume the component models to be available and that functionally-compatible components have been identified. In the next chapters, we focus on the synthesis and implementation of mediators.

Chapter 4

Automated Synthesis of Mediators

> *"When you and I speak or write to each other, the most we can hope for is a sort of incremental approach toward agreement, toward communication, toward common usage of terms."*
>
> — Douglas Lenat, researcher and CEO of Cycorp, (1950-)

To enable two functionally-compatible components to interoperate, the mediator must solve the differences between the interfaces of these components and coordinate their behaviours at both the application and middleware layers. In this chapter, we focus on the automated synthesis of mediators at the application layer as it provides the appropriate abstraction to reason about the meaning of the actions of components' interfaces using a domain specific ontology. We define the formal methodology to establish semantic correspondences between the actions of components' interfaces (i.e., interface matchings) where ontologies serve as a central reference point for translation and reconciliation of the meaning between actions. We also show how these correspondences can be efficiently computed using constraint programming. We then describe the algorithms used to generate the mediator by exploring the behaviours of the components and composing the generated interface matchings so as to guarantee that these components interact successfully. To illustrate the approach, we consider an example from the file management domain which shows the need for interoperability between a WebDAV client and the Google Docs service. This case study also illustrates the case of one-to-many correspondence between the actions of the two components, i.e., one action required from a component is translated into a se-

quence of actions provided by the other component. Furthermore, both components are based on HTTP, which allows us to focus on their differences at the application layer.

4.1 The File Management Example

The migration from desktop applications to Web-based services is scattering user files across a myriad of remote servers (e.g., Apple iCloud[1] and Microsoft Skydrive[2]). This dissemination poses new challenges for users, making it more difficult for them to organise, search, and share their files using their preferred applications. This situation, though cumbersome from a user perspective, simply reflects the way file management applications have evolved. As a result, users are forced to juggle between a plethora of applications to share their files rather than using their favourite application, regardless of the service it relies on or the standard on which it is based.

Among the standards allowing collaborative management of files, WebDAV (Web Distributed Authoring and Versioning) is an IETF specification that extends the Hypertext Transfer Protocol (HTTP) to allow users to create, read and change documents remotely. It further defines a set of properties to query and manage information about these documents, organise documents using collections, and defines a locking mechanism in order to assign a unique editor of a document at any time. Another example is represented by the Google Docs service that allows users to create, store, and search Google documents and collections, and also to share and collaborate online in the editing of these documents. These functionalities can be accessed using a Web browser or using the Google proprietary API.

Although these two systems offer similar functionalities and use HTTP as the underlying transport protocol, they are unable to interoperate. For example, a user cannot access his Google Docs documents using his favourite WebDAV client (e.g., Mac Finder) as depicted in Figure 4.1. This is mainly due to the syntactic naming of the data and operations used in WebDAV and Google Docs, together with the order according to which these operations are performed. In particular, differences exist in:

- *The names and types of input/output data and operations.* For example, resource containers are called *folders* in Google Docs and *collections* in WebDAV.

[1] http://www.apple.com/icloud/
[2] http://windows.microsoft.com/skydrive/

Google Docs also distinguishes between different types of documents (e.g., presentation or spreadsheet) whereas WebDAV considers them all as files.

- *The granularity of operations.* WebDAV provides an operation for moving files from one location to another whereas Google Docs does not. However, the move operation can be carried out using existing operations to achieve the same task, i.e., by performing the upload, download, and delete operations offered by the Google Docs service.

- *The ordering of operations.* WebDAV requires operations on files to be preceded by a lock operation and followed by an unlock operation. Google Docs does not have such a requirement. Still, it allows users to restrict or release access to a document by changing its sharing settings. Hence, although the operations of the two systems can be mapped to one another, the sequencing of operations required by WebDAV is not respected by Google Docs .

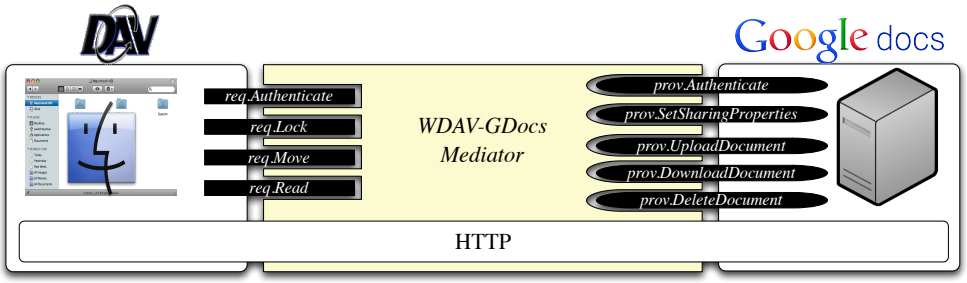

Figure 4.1. Making WebDAV client and Google Docs service interoperable

The File Management Ontology

In the interoperable file management scenario, we build upon the NEPOMUK File Ontology[1], which defines the vocabulary for describing and relating information elements and operations that are commonly used in file management applications. Figure 4.2 shows an extract of this ontology once the classification has been performed. A Resource has some ResourceProperties and is defined as the union of File and Collection. In DL, this would be written: Resource $\doteq$ File $\sqcup$ Collection. The *dot* above the "equals" symbol designates a declarative axiom. In addition, a Resource

[1] http://www.semanticdesktop.org/ontologies/nfo/

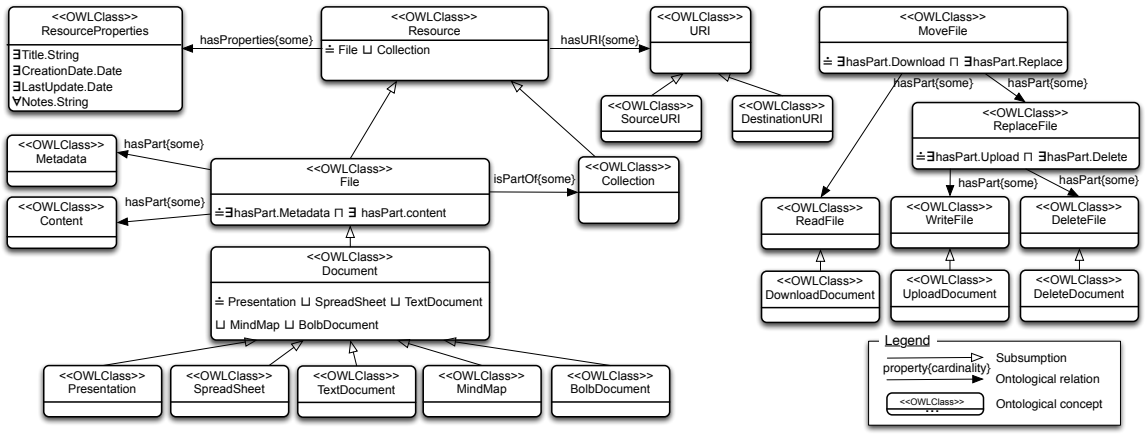

Figure 4.2. The file management ontology

concept has some resource properties: Resource $\dot{\sqsubseteq}$ ∃hasProperties.ResourceProperties. The ResourceProperties has several data properties, e.g., Title of type String. It can then be inferred that a File is a Resource: File $\sqsubseteq$ Resource and that a File also has some properties File $\sqsubseteq$ ∃hasProperties.ResourceProperties

In a similar way, when one states that MoveFile is made up of ReadFile and ReplaceFile, which in turn is made up of WriteFile and DeleteFile, the ontology reasoner can infer that MoveFile is defined as the aggregation of the three concepts, i.e., MoveFile = ReadFile $\oplus$ WriteFile $\oplus$ DeleteFile where = is equivalent to a double subsumption. Hence, by giving a formal definition to each concept, the reasoner can infer a hierarchy of concepts.

Behavioural Specifications of the File Management Components

The behavioural specification of the WebDAV client ($WDAV$) in FSP is as follows:

$$
\begin{array}{lcl}
WDAV & = & (<\text{Authenticate}, \{\text{Username}, \text{Password}\}, \{\text{Authorisation}\} >\!\!\rightarrow \text{P1}), \\
\text{P1} & = & (<\text{Lock}, \{\text{SourceURI}\}, \{\text{Acknowledgment}\} >\!\!\rightarrow \text{P2} \\
 & & \mid <\text{Logout}, \emptyset, \{\text{Acknowledgment}\} >\!\!\rightarrow \text{END}), \\
\ldots & &
\end{array}
$$

$$
\begin{array}{lcl}
\ldots & & \\
\mathsf{P2} & = & (<\mathsf{ListFolder}, \{\mathsf{SourceURI}\}, \{\mathsf{FileList}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{ReadFile}, \{\mathsf{SourceURI}\}, \{\mathsf{File}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{WriteFile}, \{\mathsf{File}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{DeleteFile}, \{\mathsf{File}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{MoveFile}, \{\mathsf{SourceURI}, \mathsf{DestinationURI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{Unlock}, \{\mathsf{SourceURI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1}).
\end{array}
$$

First, clients start an interaction by logging in. Then, before executing any operation, clients have to lock the resource, perform the desired operation and then unlock it again. Finally, they log out to terminate. The behaviour of the Google Docs service (*GDocs*) is defined as follows:

$$
\begin{array}{lcl}
\mathit{GDocs} & = & (<\overline{\mathsf{Authenticate}}, \{\mathsf{Username}, \mathsf{Password}\}, \{\mathsf{Authorisation}\} >\!\!\rightarrow \mathsf{P1}), \\
\mathsf{P1} & = & (<\overline{\mathsf{SetSharingProperties}}, \{\mathsf{SourceURI}, \mathsf{SharingProperties}\}, \\
 & & \quad \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{ListCollection}}, \{\mathsf{SourceURI}\}, \{\mathsf{DocumentList}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{DownloadDocument}}, \{\mathsf{SourceURI}\}, \{\mathsf{Document}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{UploadDocument}}, \{\mathsf{Metadata}, \mathsf{Content}, \mathsf{DestinationURI}\}, \\
 & & \quad \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{DeleteDocument}}, \{\mathsf{SourceURI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{Logout}}, \emptyset, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{END}).
\end{array}
$$

GDocs expects clients to authenticate themselves first. Then, they can list collections of documents or download, upload and delete documents. Finally, they log out to terminate.

To achieve interoperability between *WDAV* and *GDocs*, a mediator must be synthesised which solves the differences between the interfaces of these components and coordinates their behaviours in order to ensure their successful interaction. To that end, we first compute interface matchings, which specify how the actions required by *WDAV* can be performed using actions provided by *GDocs*. Note that, even though we compute the interface matching both ways, it is unnecessary to match the interface of *GDocs* against that of *WDAV* since *GDocs* does not require any action. Then, we synthesise the mediator that composes the interface matchings in a way that guarantees that the two components progress and reach their final states without errors (e.g., deadlock).

4.2 Specification of Interface Matching

To enable *WDAV* and *GDocs* to interoperate, the mediator must translate the actions required by the former into actions provided by the latter. This translation is only possible if there is a semantic correspondence between the actions required by *WDAV* and those provided by *GDocs*. Establishing the semantic correspondence between the actions of the components' interfaces is a crucial step towards the synthesis of mediators. In this section, we specify the conditions under which such a correspondence, i.e., interface matching, may be established.

Let us consider two components' interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$. Matching $\mathcal{I}_1$ with $\mathcal{I}_2$, written $Match\,(\mathcal{I}_1, \mathcal{I}_2)$, consists in finding all pairs (X_1, X_2) where $X_1 = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle, \alpha_{i=1..m} \in \mathcal{I}_1$ and $X_2 = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n} \rangle, \overline{\beta_{j=1..n}} \in \mathcal{I}_2$ such that X_1 *matches with* X_2, denoted $X_1 \mapsto X_2$, if the required actions of X_1 can be safely performed by calling the provided actions of X_2. In addition, this pair is *minimal*, that is, any other pair of sequences of actions (X_1', X_2') such that X_1' matches with X_2' would have either X_1 as a subsequence of X_1' or X_2 as a subsequence of X_2'. The interface matching is then specified as follows:

$$
\begin{array}{l}
(X_1, X_2) \in Match\,(\mathcal{I}_1, \mathcal{I}_2) = \\
\{\ (X_1, X_2) | \\
\quad X_1 = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle, \alpha_{i=1..m} \in \mathcal{I}_1 \\
\wedge\, X_2 = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n} \rangle, \overline{\beta_{j=1..n}} \in \mathcal{I}_2 \\
\wedge\, X_1 \mapsto X_2 \\
\wedge\, \nexists\, (X_1', X_2') \mid\ X_1' = \langle \alpha_1, \alpha_2, \ldots, \alpha_{m'} \rangle,\ \alpha_{i=1..m'} \in \mathcal{I}_1 \\
\qquad\qquad \wedge\, X_2' = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_{n'}} \rangle, \overline{\beta_{j=1..n'}} \in \mathcal{I}_2 \\
\qquad\qquad \wedge\, (X_1' \mapsto X_2') \\
\qquad\qquad \wedge\ (m' < m) \wedge (n' < n) \\
\}
\end{array}
$$

Likewise, $Match\,(\mathcal{I}_2, \mathcal{I}_1)$ represents the set of all pairs (X_2, X_1), where X_2 is a sequence of required actions of $\mathcal{I}_2$ and X_1 is a sequence of provided actions of $\mathcal{I}_1$, such that X_2 matches with X_1 and this matching is minimal.

Let $X_1 = \langle \alpha_1, \ldots, \alpha_m \rangle$ be a sequence of required actions and $X_2 = \langle \overline{\beta_1}, \ldots, \overline{\beta_n} \rangle$ be a sequence of provided actions. To facilitate the definition of a matching of X_1 with X_2, we consider the following cases:

- $m = 1$ *and* $n = 1$. This is a one-to-one matching, denoted $X_1 \overset{1-1}{\longmapsto} X_2$, and it states that an action required by one component can be safely performed by an action provided by the other component. For example, the $<\mathsf{ReadFile}, \{\mathsf{SourceURI}\}, \{\mathsf{File}\}>$ action required by *WDAV* can be performed using the $<\overline{\mathsf{DownloadDocument}}, \{\mathsf{SourceURI}\}, \{\mathsf{Document}\}>$ action provided by *GDocs*.

- $m = 1$ and $n \geq 1$. This is a one-to-many matching, denoted $X_1 \overset{1-n}{\longmapsto} X_2$, and refers to action split/merge, i.e., when an action required by one component is provided by a sequence of actions from the other. For example, the $<\mathsf{MoveFile}, \{\mathsf{SourceURI}, \mathsf{DestinationURI}\}, \{\mathsf{Acknowledgment}\}>$ action required by *WDAV* can be performed using the $<\overline{\mathsf{DownloadDocument}}, \{\mathsf{SourceURI}\}, \{\mathsf{Document}\}>$, $<\overline{\mathsf{UploadDocument}}, \{\mathsf{Metadata}, \mathsf{Content}, \mathsf{DestinationURI}\}, \{\mathsf{Acknowledgment}\}>$, and $<\overline{\mathsf{DeleteDocument}}\ , \{\mathsf{SourceURI}\}, \{\mathsf{Acknowledgment}\}>$ actions provided by *GDocs*.

- $m \geq 1$ and $n \geq 1$. This is the most general case and refers to a many-to-many matching, denoted $X_1 \overset{m-n}{\longmapsto} X_2$. It is used to specify the case where one sequence of actions corresponds to another sequence of actions.

Each matching is associated with a process that specifies how the sequences of actions are coordinated. In the following, we specify the conditions that must be satisfied by sequences of actions in order to match. We first give a formal definition in the one-to-one case, which we extend to the one-to-many and many-to-many cases. Note that we do not distinguish between the aggregation ($\oplus$) and disjunction ($\sqcup$) constructors when computing interface matchings as they both represent compositions of concepts. The distinction is however maintained at the ontological level, and also for data translations: in the case of disjunction $E \doteq C \sqcup D$, the translation consists in producing an instance of E by assigning to it either an instance of C or an instance of D. While in the case of aggregation $E \doteq C \oplus D$, an instance of E is produced by combining its parts, i.e., both C and D instances, in the appropriate way.

4.2.1 One-to-One Matching

A required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ matches with a provided action $\overline{\beta} = \langle \overline{b}, I_b, O_b \rangle \in \mathcal{I}_2$, written $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$, iff:

1. $b \sqsubseteq a$
2. $I_a \sqsubseteq I_b$
3. $I_a \sqcup O_b \sqsubseteq O_a$

The idea behind this matching is that a required action can be achieved using a provided one if the former supplies the required input data while the latter provides the needed output data, and the required operation is less demanding than the provided one. This coincides with the Liskov Substitution Principle [LW94] where ontological subsumption can be used in ways similar to type subsumption.

As a result, we generate the matching process M_{1-1} that synchronises with each component by executing its dual action in order to let the two components progress as depicted in Figure 4.3. Hence, the one-to-one matching process corresponding to $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$ is defined as follows: $M_{1-1}(\alpha, \overline{\beta}) = (\beta \rightarrow \overline{\alpha} \rightarrow \mathsf{END})$.

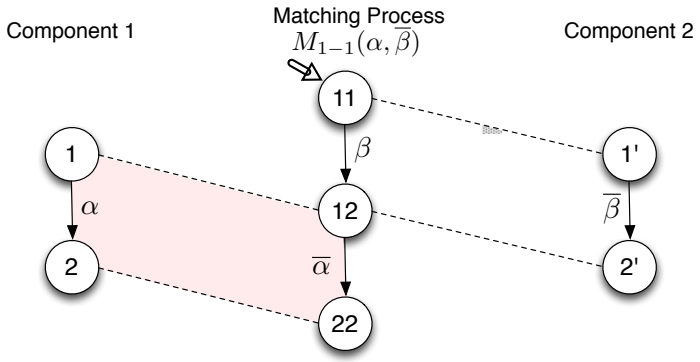

Figure 4.3. One-to-one matching process: $M_{1-1}(\alpha, \overline{\beta})$

Let us consider the <ReadFile, {SourceURI}, {File}> action required by *WDAV* and the <$\overline{\text{DownloadDocument}}$, {SourceURI}, {Document}> action provided by *GDocs*. We can verify from the file management ontology in Figure 4.2 that:

1. DownloadDocument $\sqsubseteq$ ReadFile,

2. SourceURI $\sqsubseteq$ SourceURI since subsumption is reflexive, and

3. Document $\sqsubseteq$ File.

As a result, we generate a corresponding matching process M_{1-1}(ReadFile, $\overline{\text{DownloadDocument}}$)=(<DownloadDocument,{SourceURI}, {Document}>$\rightarrow$<$\overline{\text{ReadFile}}$, {SourceURI}, {File}>$\rightarrow$END). M_{1-1}(ReadFile, $\overline{\text{DownloadDocument}}$) requires DownloadDocument and provides ReadFile. Therefore, it needs to: (i) receive the URI from *WDAV*, (ii) transform it into that expected by DownloadDocument, (iii) send the translated URI to *GDocs*, (iv) receive the appropriate document, (i) transform it into a file, and (iv) send it back to *WDAV*.

4.2.2 One-to-Many Matching

Following the same idea, a required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ matches with a sequence of provided actions $X_2 = \left\langle \overline{\beta_i} = \left\langle \overline{b_i}, I_{b_i}, O_{b_i} \right\rangle \in \mathcal{I}_2 \right\rangle_{i=1..n}$, written $\alpha \overset{1-n}{\longmapsto} \left\langle \overline{\beta_1}, \ldots, \overline{\beta_n} \right\rangle$, iff:

1. $\bigsqcup_{i=1}^{n} b_i \sqsubseteq a$
2. $I_a \sqsubseteq I_{b_1}$
3. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{i-1} O_{b_j} \right) \sqsubseteq I_{b_i}$
4. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{n} O_{b_j} \right) \sqsubseteq O_a$

The first condition states that the operation a can be appropriately performed using b_i operations, that is, the disjunction of all b_i is subsumed by a. The second ensures that the sequence of provided actions can be initiated since the input data of the first action I_{b_1} can be obtained from the input data of the required action I_a. The third condition specifies that the input data of each action can be produced from the data previously received either as input from α or as output from the preceding β_j $(j < i)$. This is necessary since an action can only be executed if its input data is available. The fourth condition guarantees that the required output data O_a can be

obtained from the set of data accumulated during the execution of all the provided actions.

As a result, we generate the matching process $M_{1-n}(\alpha, X_2)$, depicted in Figure 4.4, which consumes all the provided actions so as to provide the action $\overline{\alpha}$. The matching process corresponding to $\alpha \overset{1-n}{\longmapsto} \langle \overline{\beta_1}, \ldots, \overline{\beta_n} \rangle$ is as follows: $M_{1-n}(\alpha, X_2) = (\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_n \rightarrow \overline{\alpha} \rightarrow \mathsf{END})$.

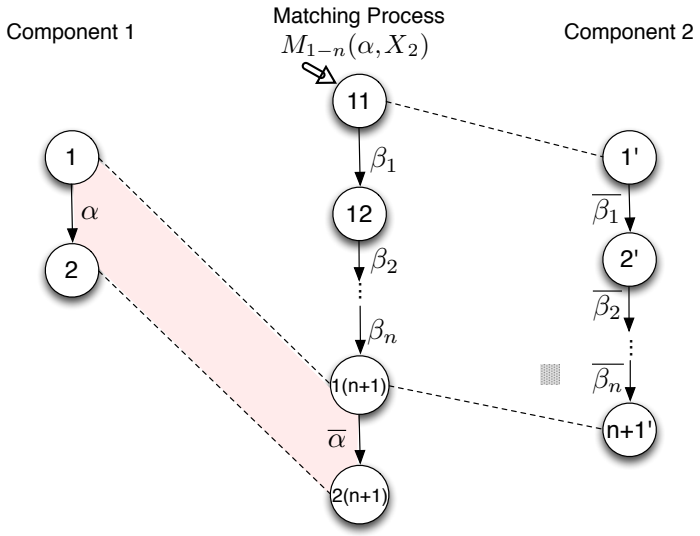

Figure 4.4. One-to-many matching process: $M_{1-n}(\alpha, X_2)$

Let us consider the <MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}> action required by *WDAV* and the three actions <$\overline{\text{DownloadDocument}}$, {SourceURI}, {Document}>, <$\overline{\text{UploadDocument}}$, {Metadata, Content, DestinationURI}, {Acknowledgment}>, and <$\overline{\text{DeleteDocument}}$, {SourceURI}, {Acknowledgment}> provided by *GDocs*. First, the condition on the semantics of the operations is verified, that is, DownloadDocument $\oplus$ UploadDocument $\oplus$ DeleteDocument $\sqsubseteq$ MoveFile. Then, both DownloadDocument and the DeleteDocument can be performed as they only expect a SourceURI as input, which is produced by MoveFile. UploadDocument requires input data, which can only be provided by DownloadDocument since Document $\sqsubseteq$ Metadata $\oplus$ Content (see Figure 4.2) and can only be executed after the UploadDocument operation. Hence, we can have the following matching:

<MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}>
↦<<$\overline{\text{DownloadDocument}}$, {SourceURI}, {Document}>
<$\overline{\text{UploadDocument}}$, {Metadata, Content, DestinationURI},{Acknowledgment}>,
<$\overline{\text{DeleteDocument}}$, {SourceURI}, {Acknowledgment}>>

We can generate a corresponding matching process M_{1-n}(MoveFile, <$\overline{\text{DownloadDocument}}$, $\overline{\text{UploadDocument}}$, $\overline{\text{DeleteDocument}}$>), which: (i) receives the URI from the WebDAV client, (ii) transforms it into that expected by the DownloadDocument action using a translation function f_1, (iii) sends the translated URI to the Google Docs service, (iv) receives the appropriate document, (v) builds the parameters of the UploadDocument action by transforming the received document and the destination URI, (vi) sends the parameters and receives the acknowledgment, (vii) creates the source URI expected by the DeleteDocument, (viii) sends it back to the Google Docs service, (viii) receives the corresponding acknowledgment, (ix) generates the acknowledgment expected by the MoveFile action, and (x) sends it back to the WebDAV client.

Note that since there is no data dependency between the upload and delete actions, there exists another matching using the same actions in a different order:

<MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}>
↦<<$\overline{\text{DownloadDocument}}$, {SourceURI}, {Document}>
<$\overline{\text{DeleteDocument}}$, {SourceURI}, {Acknowledgment}>,
<$\overline{\text{UploadDocument}}$, {Metadata, Content, DestinationURI}, {Acknowledgment}>>

4.2.3 Many-to-Many Matching

In the same vein, a sequence of required actions $X_1 = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ matches with a sequence of provided actions $X_2 = \left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{j=1..n}$, written $\langle \alpha_1, \ldots, \alpha_l, \ldots, \alpha_m \rangle \overset{m-n}{\longmapsto} \left\langle \overline{\beta_1}, \ldots, \overline{\beta_n} \right\rangle$, iff:

1. $\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i$
2. $\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}$
3. $\left(\bigsqcup_{j=1}^{l} I_{a_j} \right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h} \right) \sqsubseteq I_{b_i}$

4. $\forall h \in [1, l[,\ O_{a_h} = \emptyset$

5. $\forall h \in [l, m], \left(\bigsqcup_{i=1}^{h} I_{a_i}\right) \bigsqcup \left(\bigsqcup_{k=1}^{n} O_{b_k}\right) \sqsubseteq O_{a_h}$

The first condition states that the a_i operations can be appropriately performed using b_j operations. The second condition states that the execution of provided actions can be initiated if the necessary input data I_{b_1} can be computed based on the data previously received. The third ensures that the input data of each action can be acquired by considering the received input and the output of the preceding actions. Since we assume synchronous semantics, a required action can only be achieved if its output is available, and analogously a provided action can be executed only if its input is available. However, we can accumulate the data produced by required actions and allow them to progress if they do not require any output. Hence, the fourth condition specifies that the first $l-1$ actions do not require any output and can be executed before the provided actions. Finally, the last condition states that the output of the remaining required actions, i.e., from l to m, can be ascertained by taking into account the previous inputs as well as the outputs generated by the provided actions.

Consequently, we generate the matching process $M_{m-n}(X_1, X_2)$ depicted in Figure 4.5. $M_{m-n}(X_1, X_2)$ first synchronises with the $l-1$ first required actions since no output data is necessary for them to be executed. But only after consuming all provided actions, can $M_{m-n}(X_1, X_2)$ synchronise with the subsequent required actions —from l to m— since the output data necessary for their achievement are produced by the provided actions. The matching process corresponding to $\langle \alpha_1, \ldots, \alpha_l, \ldots, \alpha_m \rangle \overset{m-n}{\longmapsto} \langle \overline{\beta_1}, \ldots, \overline{\beta_n} \rangle$ is as follows: $M_{m-n}(X_1, X_2) = (\overline{\alpha_1} \to \cdots \to \overline{\alpha_{l-1}} \to \beta_1 \to \cdots \to \beta_n \to \overline{\alpha_l} \to \overline{\alpha_{l+1}} \cdots \to \overline{\alpha_m} \to \mathsf{END})$.

To sum up, the conditions state that: (i) the functionality offered by the provided actions covers that of the required actions, (ii) each provided action has its input data available (in the right format) at the time of execution, and (iii) each required action has its output data available (also in the appropriate format) at the time of execution. Even though these matchings do not cover every possible mismatch —this would mean that we are able to prove computational equivalence— they cover a large enough set of mismatches with respect to practical and real case studies and other automated approaches for inferring interface matchings.

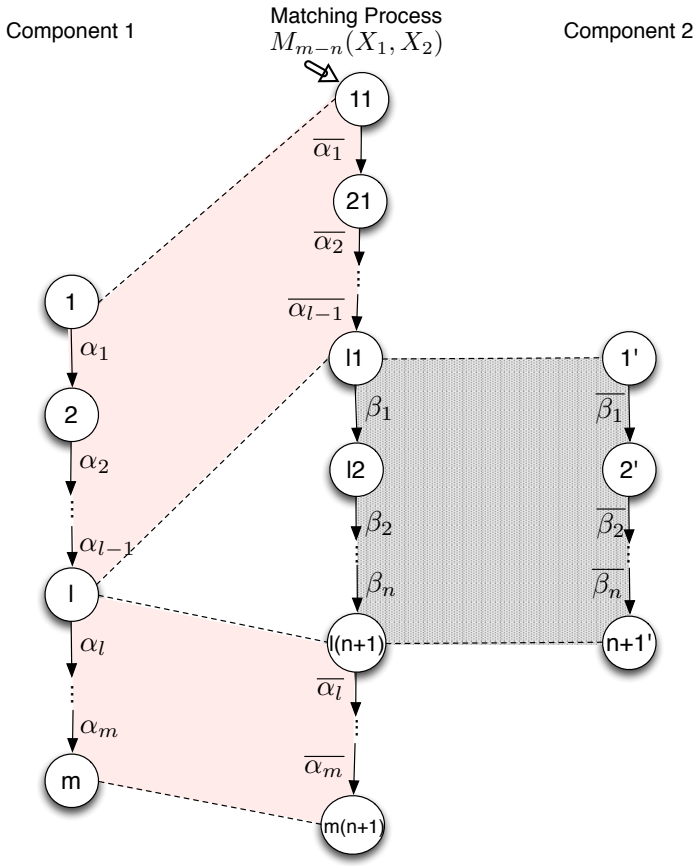

Figure 4.5. Many-to-many matching process: $M_{m-n}(X_1, X_2)$

4.3 Computation of Interface Matching using Constraint Programming

Matching interface $\mathcal{I}_1$ to interface $\mathcal{I}_2$ consists in searching, among all the possible pairs of sequences of required actions of $\mathcal{I}_1$ and sequences of provided actions of $\mathcal{I}_2$, those that verify the matching conditions specified in the previous section. Furthermore, each pair is minimal in the sense that is made up of the smallest sequences of actions verifying the matching conditions. We prove that interface matching is an NP-complete problem and use Constraint Programming (CP) [RVBW06] to deal with it effectively.

Many arithmetical and logical operators are managed by existing CP solvers. However, although there are some attempts to integrate ontologies with CP [JM03,

Lab03], none supports ontology-related operators such as subsumption or disjunction of concepts. In order to use CP to compute interface matching, we need to enable ontology reasoning within CP solvers. Therefore, we propose to represent the ontological relations we are interested in using arithmetic operators supported by existing solvers. In particular, we devise an approach to associate a unique code to each ontological concept and where disjunction and subsumption relations amount to boolean operations.

In the following, we begin by proving that interface matching is an NP-complete problem. We formulate the interface matching as a constraint satisfaction problem and show how CP can be used to solve it efficiently. We further propose an ontology encoding that considers subsumption and union in order to translate ontology reasoning into finite-domain constraints and thereby speed up the reasoning at runtime.

4.3.1 Complexity of Interface Matching

We prove that interface matching is NP-complete using polynomial-time reductions from the set cover problem [Kar72]. We recall that in the set cover problem, we are given a set of n elements U and a finite family of its subsets $C = \{S_1, \ldots, S_m\}$ such that $S_i \subseteq U$ and $\bigcup_{i=1}^{m} S_i = U$, and we must find a smallest collection of these subsets whose union is U, i.e., a family of subsets $C' = \{T_1, \ldots, T_k\}$ such that $T_j \subseteq C$ and $\bigcup_{j=1}^{k} T_i = U$.

The first step is to transform an instance of the set cover problem into an instance of interface matching. We start by building an ontology made up of disjoint concepts, each of which represents an element of U. Then, the first interface includes a unique required action whose operation is the disjunction of all the ontology concepts representing elements of U and input and output data are void. $\mathcal{I}_1 = \{< \bigsqcup_{x \in U} x, \emptyset, \emptyset >\}$. The second interface $\mathcal{I}_2$ is made up only of provided actions. Each action $\overline{\beta_i}$ is associated with a subset $S_i \in C$ and where the operation of $\overline{\beta_i}$ is the disjunction of the ontology concepts representing S_i's elements and its input and output data are void. Hence, $\mathcal{I}_2 = \bigcup_{S_i \in C} \{< \overline{\bigsqcup_{s \in S_i} s}, \emptyset, \emptyset >\}$.

Since the input and output data are void, interface matching specifies the pairs $(\alpha, \overline{\beta_1} \ldots \overline{\beta_k}) \in \mathcal{I}_1 \times (\mathcal{I}_2)^k$ verifying $\bigsqcup_{i=1}^{k} \left(\bigsqcup_{s \in S_i} s \right) \sqsubseteq \bigsqcup_{x \in U} x$. To get a solution to the set

cover, it suffices to pick the subsets associated with the shortest sequence, which can be performed in polynomial time. Therefore, interface matching is NP-hard.

Further, we can verify the subsumption relations between a pair of sequences of actions in polynomial time, we can state that interface matching computation is NP-complete.

4.3.2 Interface Matching as a Constraint Satisfaction Problem

Constraint programming is the study of combinatorial problems by stating constraints (conditions, qualities) which must be satisfied by the solution(s) [RVBW06]. These problems are defined as a Constraint Satisfaction Problem and modelled as a triple (X, D, C):

- *Variables*: $X = \{x_1, x_2, ..., x_n\}$ is the set of variables of the problem.
- *Domains*: D is a function which associates to each variable x_i its domain $D(x_i)$, i.e., the set of possible values that can be assigned to x_i.
- *Constraints*: $C = \{C_1, C_2, ..., C_m\}$ is the set of constraints. A constraint C_j is a mathematical relation defined over a subset $x^j = \{x_1^j, x_2^j, \ldots, x_{n^j}^j\} \subseteq X$ of variables, which restricts their possible values. Constraints are used actively to deduce unfeasible values and delete them from the domains of variables. This mechanism is called *constraint propagation*. Efficient algorithms specific to each constraint are used in this propagation.

Solving a constraint satisfaction problem consists in finding the tuple (or tuples) $v = (v_1, ..., v_n)$ where $v_i \in D(x_i)$ and such that all constraints C_j are satisfied. Thus, CP uses constraints to state the problem declaratively without specifying a computational procedure to enforce them. The latter task is carried out by a solver. The constraint solver implements intelligent search algorithms such as backtracking and branch and bound which are exponential in time in the worst case, but may be very efficient in practice. They also exploit the arithmetic properties of the operators used to express the constraint to quickly check, discredit partial solutions, and prune the search space substantially.

We represent interface matching $Match\,(\mathcal{I}_1, \mathcal{I}_2)$ as a constraint satisfaction problem below:

- *Variables*: $X = \{x_1, x_2\}$ where x_1 represents a sequence of required actions of $\mathcal{I}_1$ and x_2 represents a sequence of provided actions of $\mathcal{I}_2$.
- *Domains*: x_1 and x_2 can be any ordering of any length of the actions of $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. Hence, $D(x_1) = \bigcup_{k=1}^{|\mathcal{I}_1|} P_k(\mathcal{I}_1)$ and $D(x_2) = \bigcup_{k=1}^{|\mathcal{I}_2|} P_k(\mathcal{I}_2)$ where $P_k(S)$ denotes the set of k-permutations of the elements of the set S. Indeed, x_1 is a sequence of actions (hence the permutations) of $\mathcal{I}_1$ of length k varying between 1 and the cardinality of $\mathcal{I}_1$, i.e, $1 < k < |\mathcal{I}_1|$. This is also the case for x_2.
- *Constraints*: the constraints are defined by the matching conditions specified in Section 4.2. As for the minimality of the matching, we eliminate interchangeable solutions [Fre91] using the *subsequence relation*, which is a partial order relation defined on the domain of each variable. Let us consider two sequences of actions $A = <a_i>_{i=1..m}$ and $B = <b_j>_{j=1..n}$. A is a subsequence of B iff $m < n$ and $\forall k \in 1..m$, $a_k = b_k$. Exploiting this partial order relation for the branch and bound algorithm used to solve the CSP allows us to keep only minimal matching verifying the rest of the constraints.

The solutions of the constraint satisfaction problem are the smallest, according to the subsequence relation, pairs of action sequences $(\alpha, \overline{\beta}) \in D(x_1) \times D(x_2)$ such that the required actions of α can be safely achieved using the provided actions of $\overline{\beta}$.

4.3.3 Leveraging Constraint Programming for Ontological Reasoning

Since none of the existing CP solvers supports ontology reasoning, our goal is to use the arithmetical and logical operators supported by these solvers in order to represent ontological relations. To do so, we define a bit vector encoding of the ontology, which is correct and complete regarding the subsumption and disjunction axioms. Correctness means that if the encoding asserts that a concept subsumes another concept or that a concept is a disjunction of other concepts, then these relations can be verified in the ontology. Completeness signifies that the subsumption and the disjunction relations specified in the ontology can be verified by the encoding. Specifically, we define the relations $\mathcal{R}_{\sqsubseteq}$ and $\mathcal{R}_{\sqcup}$ such that:

$$C \sqsubseteq D \iff \mathcal{R}_{\sqsubseteq}(C, D)$$
$$E = C \sqcup D \iff \mathcal{R}_{\sqcup}(E, C, D)$$

Since we do not distinguish between the aggregation ($\oplus$) and disjunction ($\sqcup$) constructors when computing the interface matching, the aggregation relation is also encoded using the $\mathcal{R}_{\sqcup}$ relation as follows:

$$E = C \oplus D \Longleftrightarrow \mathcal{R}_{\sqcup}(E, C, D)$$

The algorithm for encoding an ontology (Algorithm 1) takes the classified ontology as its input, i.e., an ontology that also includes inferred axioms, and returns a map

Algorithm 1 : Ontology Encoding

Require: Classified ontology$\mathcal{O}$
Ensure: $Code[]$: maps each concept $C \in \mathcal{O}$ to a bit vector

for all $C \in Concepts(\mathcal{O})$ **do**
  $Set[C] \leftarrow \{\text{NewElement}()\}$
end for
for all $C \in Concepts(\mathcal{O})$ **do**
  for all $Des \in \text{Descendants}(C)$ **do**
    $Set[C] \leftarrow Set[C] \cup Set[Des]$
  end for
end for
$disjunctionAxiomList = \text{Sort}(DisjunctionAxioms(\mathcal{O}))$
for all $A \doteq \bigsqcup_{i=1}^{n} A_i \in disjunctionAxiomList$ **do**
  $\mathcal{D} \leftarrow Set[A] \setminus \bigcup_{i=1}^{n} Set[A_i]$
  for all $d \in \mathcal{D}$ **do**
    $Set[A] \leftarrow Set[A] \setminus \{d\}$
    for all A_i **do**
      $d_i \leftarrow \{NewElement()\}$
      $Set[A_i] \leftarrow Set[A_i] \cup \{d_i\}$
      for all $Asc \in Ascendants(A_i)$ **do**
        $Set[Asc] \leftarrow Set[Asc] \cup d_i$
      end for
    end for
    for all $Des \in \text{Descendants}(A) \mid d \in Set[Des]$ **do**
      $Set[Des] \leftarrow (Set[Des] \setminus \{d\}) \bigcup \left(\bigcup_{i=1}^{n} d_i \right)$
    end for
  end for
end for
$Code[] \leftarrow SetsToBitVectors(Set[])$
return $Code[]$

that associates each concept with a bit vector. We first use sets to encode the ontology concepts such that subsumption coincides with set inclusion and disjunction with set union. Then, we represent the sets using bit vectors whose size is the number of elements of all sets. Each bit is set to 1 if the corresponding element belongs to the set and to 0 otherwise. The type of elements of the sets does not matter, they are just temporary objects used to perform the encoding.

The first step of the encoding algorithm is to assign a unique element to the set that represents each concept (Lines 1−3). Then, we augment the set of each concept with the elements of the sets associated with the concepts it subsumes, i.e., its descendants (Lines 4−8) since subsumption essentially comes down to set inclusion of the instances of concepts.

We then move to disjunction axioms. We sort the axioms so that each element is made up of simple concepts or preceding concepts in the list (Line 9). For each disjunction axiom $A \doteq \bigsqcup_{i=1}^{n} A_i$, we consider the set $\mathcal{D}$ of elements that belong to the set representing A but which are not included in any of the sets of its composing classes A_i (Line 11). These elements are either the distinguishing element of A, or put into A's set by one of its sub-concepts during the previous step. The latter case represents the case where a concept is subsumed by the disjunction A but not by any of its individual concepts. To preserve the subsumption, each element $d \in \mathcal{D}$ is divided into n elements, each of which is added to one of the composing classes $A_{i=1..n}$. Hence, we first remove d from A (Line 13). Then, we create a new element d_i and add it to A_i's set as well as to the sets of its subsuming concepts, which include A (Lines 14−20). We also replace the element d in A's descendants by the new elements it was divided into (Lines 21−23). Finally, we encode the sets using bit vectors where each bit indicates whether or not an element belongs to the set (Line 26).

As a result, subsumption can be performed using bitwise AND as follows:

$$C \sqsubseteq D \iff Code[C] \wedge Code[D] = Code[C]$$

and corresponds to the $\mathcal{R}_{\sqsubseteq}$ relation we were looking for. The $\mathcal{R}_{\sqcup}$ relation corresponding to disjunction is represented by bitwise OR:

$$C = \bigsqcup_{i=1}^{n} A_i \iff Code[A] = \bigvee_{i=1}^{n} Code[A_i]$$
$$C = \bigoplus_{i=1}^{n} A_i \iff Code[A] = \bigvee_{i=1}^{n} Code[A_i]$$

Example. Let us consider the extract of the file management ontology depicted in Figure 4.6. File subsumes Document which is defined as the disjunction of Presentation, SpreadSheet, and TextDocument. During the first step, we associate an element, which we represent as a natural number, to each concept and put it into its ascendants. The element *'1'* represents the bottom concept $\bot$ subsumed by all concepts. Then, we consider the Document $\doteq$ Presentation $\sqcup$ SpreadSheet $\sqcup$ TextDocument disjunction. The *'5'* element belongs to Document but not to any of its composing concepts, so we split it into three elements (*'51'*, *'52'*, and *'53'*) and assign each of them to the composing elements and to all its ascendants during step 2. Then during step 3, we encode sets as bit vectors. For example, Presentation includes 1 at the position of *'1'*, *'2'*, and *'51'* elements; and 0 at all other positions. The bitwise AND between the codes of File and Document corresponds to the code of Document ($11111111 \wedge 1111111 = 1111111$), which is equivalent to stating that File subsumes Document. We can then write $\mathcal{R}_{\sqsubseteq}$(Document, File). The bitwise OR between the codes of Presentation, SpreadSheet, and TextDocument is 1111111, which corresponds to the value of Document. We can then write $\mathcal{R}_{\sqcup}$(Document, Presentation, SpreadSheet, TextDocument).

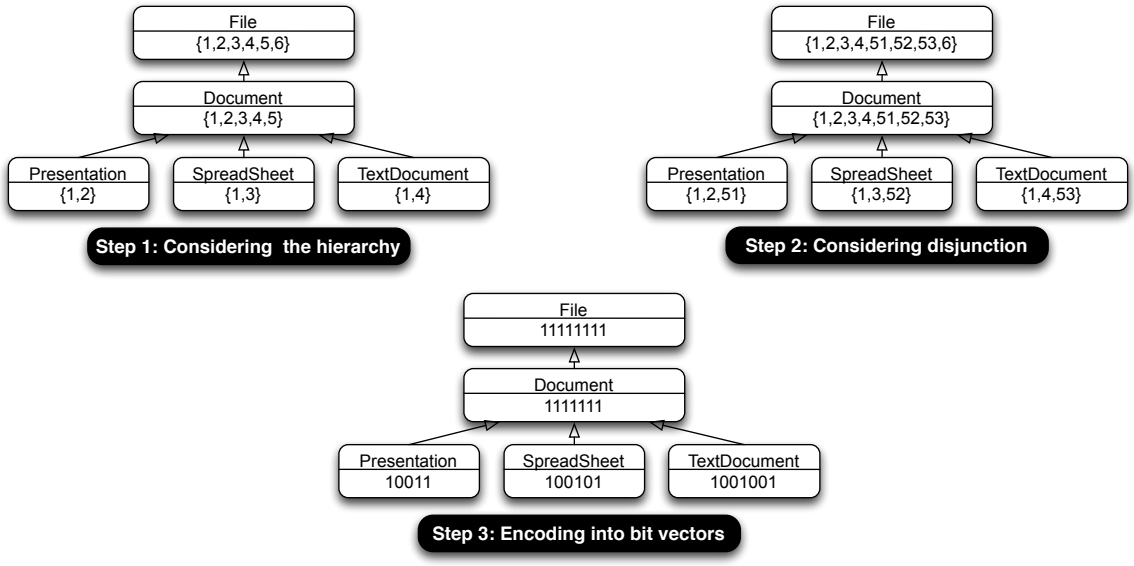

Figure 4.6. Illustrating ontology encoding on an extract of the file ontology

The encoded ontology is used by the CP solver when computing the interface matching to check if a constraint is verified. For

example, to match <ReadFile, {SourceURI}, {File}> from $\mathcal{I}_{WDAV}$ with <$\overline{\text{DownloadDocument}}$, {SourceURI}, {Document}> from $\mathcal{I}_{GDocs}$, the CP solver verifies the subsumption between output data, i.e., File should subsume Document. This verification is performed using the code associated with each concept.

4.4 Synthesising Correct-by-Construction Mediators

To enable functionally-compatible components to interoperate, the mediator must not only solve the differences between their interfaces but also coordinate their behaviours in order to ensure their correct interaction. Hence, given $Match\,(\mathcal{I}_1, \mathcal{I}_2)$ and $Match\,(\mathcal{I}_2, \mathcal{I}_1)$, where every required action is involved in at least one matching, we must either generate a mediator M that composes the associated matching processes in order to allow both components to interact correctly, or determine that no such mediator exists. Assuming that the components' behaviours are represented using P_1 and P_2 processes, to allow the components to interact correctly, the mediator M must coordinate their behaviours so as to ensure that the parallel composition $P_1 \| M \| P_2$ successfully terminates by reaching an END state. If a mediator exists, then we say that P_1 and P_2 *are behaviourally compatible through a mediator* M, written $P_1 \leftrightarrow_M P_2$.

We incrementally build a mediator M by forcing the two processes P_1 and P_2 to progress consistently so that if one requires the sequence of actions X_1, the other process is ready to engage in a sequence of provided actions X_2 with which X_1 matches. Given that an interface matching guarantees the semantic compatibility between the actions of the two components, then the mediator synchronises with both processes and compensates for the differences between their actions by performing the necessary translations. This is formally described as follows:

$$\begin{array}{ll} \text{if} & P_1 \overset{X_1}{\Rightarrow} P_1' \text{ and } \exists\,(X_1, X_2) \in Match\,(\mathcal{I}_1, \mathcal{I}_2) \\ \text{such that} & P_2 \overset{X_2}{\Rightarrow} P_2' \text{ and } P_1' \leftrightarrow_{M'} P_2' \\ \text{then} & P_1 \leftrightarrow_M P_2 \text{ where } M = M_{m-n}(X_1, X_2); M' \end{array}$$

Similarly, in the other direction:

$$\begin{array}{ll} \text{if} & P_2 \overset{X_2}{\Rightarrow} P_2' \text{ and } \exists(X_2, X_1) \in Match\,(\mathcal{I}_2, \mathcal{I}_1) \\ \text{such that} & P_1 \overset{X_1}{\Rightarrow} P_1' \text{ and } P_2' \leftrightarrow_{M'} P_1' \\ \text{then} & P_1 \leftrightarrow_M P_2 \text{ where } M = M_{m-n}(X_2, X_1); M' \end{array}$$

The mediator further consumes the extra provided actions so as to allow processes to progress. Extra provided actions are actions offered by one component that are not required by the other but need to be invoked to allow the component to continue its interaction. As long as the input data necessary to invoke the action is available, the mediator can call it and ignore the output produced. On the other hand, we do not handle extra required actions because that would involve offering some functionality, which the mediator cannot handle by itself. The support of extra provided actions is specified as follows:

$$\begin{aligned} &\text{if} \quad P_1 \xrightarrow{\overline{\beta}} P_1', \text{ and } \exists P_2 \text{ such that } P_1' \leftrightarrow_{M'} P_2 \\ &\text{then } P_1 \leftrightarrow_M P_2 \text{ where } M = (\beta \rightarrow \mathsf{END}); M' \end{aligned}$$

$$\begin{aligned} &\text{if} \quad P_2 \xrightarrow{\overline{\beta}} P_2', \text{ and } \exists P_1 \text{ such that } P_2' \leftrightarrow_{M'} P_1 \\ &\text{then } P_1 \leftrightarrow_M P_2 \text{ where } M = (\beta \rightarrow \mathsf{END}); M' \end{aligned}$$

Finally, when both processes terminate, i.e., reach an END state, then the mediator also terminates:

$$\mathsf{END} \leftrightarrow_{\mathsf{END}} \mathsf{END}$$

Note that the interface matching is not necessarily a function since an action (or a sequence of actions) can be matched with different actions (or sequences of actions). In the following, we present various cases the synthesis algorithm has to deal with (see Figure 4.7). Most of these cases are encountered within the case studies we present in Chapter 6. These cases, although simple, serve to give an intuitive notion of how to synthesise the mediator.

Case 1: Ambiguous interface matching but only one matching is applicable at a given state. Let us consider that a required action b_1 can be matched with either $\overline{a_2}$ or $\overline{c_2}$. When both processes are at their initial states (1 and $1'$ respectively), the only applicable matching is $b_1 \mapsto \overline{a_2}$ since P_2 is only able to perform this action. After applying this matching, P_1 moves to state 4, P_2 to state $2'$, and we can create a partial trace of the mediator from 11 to 42. Then, P_2 requires d_2, which matches with the provided action $\overline{c_1}$ and in which P_1 can engage. The result is that P_1 moves to state 5 and P_2 to $3'$, which are both final states. Consequently, we validate the

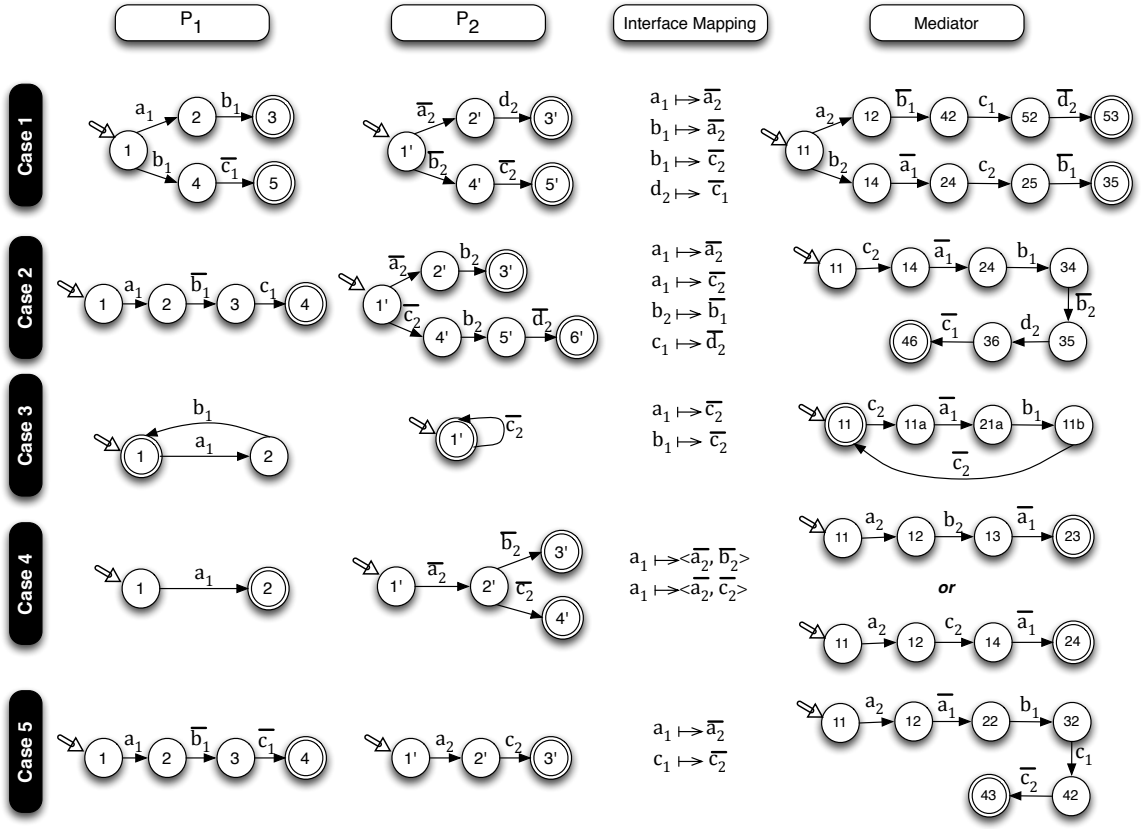

Figure 4.7. Representative cases for mediator synthesis

constructed trace and make state 53 final. We then continue exploring the other branch; a_1 only matches with $\overline{a_2}$, which leads P_1 to state 2, P_2 to $4'$ and the partial mediator to 24. Then, b_1 is again required by P_1 but at this point, it matches with $\overline{c_2}$. Finally, the two processes reach their final states and we can validate the mediator since we successfully explored all outgoing transitions with required actions.

Case 2: Ambiguous interface matching, multiple matchings applicable at a given state but only one leading to a final state. In Case 1, although an action can match with two actions, only one matching was applicable at a given state. In Case 2, the required action a_1 matches with both $\overline{a_2}$ and $\overline{c_2}$. Let us assume that the former matching is selected. P_1 and P_2 move to states 2 and $2'$ respectively, then to 3 and $3'$ after applying the $c_1 \mapsto \overline{a_2}$ matching. However, P_1 requires an action c_1 whereas P_2 reaches its final state. Consequently, we have to backtrack and select

an alternative matching. At the previous step, only $b_2 \mapsto \overline{b_1}$ was applicable but at the initial state, the $a_1 \mapsto \overline{c_2}$ matching has not been tested. We select this matching and continue the exploration until we reach states 4 and 6′, which are the final states of both processes. Hence, it is crucial in each state to keep track of the matchings that have been examined since for each outgoing transition with a required action, we need to select the appropriate matching that enables both processes to reach their final states.

Case 3: Ambiguous interface matching, multiple matchings applicable at a given state and all are valid. In this case, two matchings are both valid and allow the processes to reach their final states. However, we need to select only one of them to generate the mediator since the mediator cannot make a non-deterministic choice on the actions to invoke. The selection of the matching to use may be motivated by some non-functional property or the length of the matching but, for instance, let us assume that we select the first valid matching. Indeed, we regard the functional concerns as paramount and non-functional concerns as secondary. The result is that a correct mediator is not unique. Hence, there are two possible valid mediators: the first translates a_1 to the sequence $\overline{a_2} \rightarrow \overline{b_2}$ while the second translates a_1 to $\overline{a_2} \rightarrow \overline{c_2}$.

Case 4: Multiple required actions matching with the same provided action. In the previous cases a required action is involved in different matchings. In Case 4, a provided action is involved in different matchings: both a_1 and b_1 match with $\overline{c_2}$. After performing the a_1 to $\overline{c_2}$ matching, P_1 moves to a new state that needs to be explored as well, while P_2 returns to its initial state. After the second matching $b_1 \mapsto \overline{c_2}$, P_1 also returns to its initial state but all the outgoing transitions have been treated and it is also a final state, so the mediator is validated.

Case 5: Extra provided action. In the previous cases, the mediator was created so as to coordinate the actions required by one component with the actions provided by the other. The underlying assumption is that the mediator is not able to provide actions itself (only components do). The mediator may however consume extra provided actions in order to allow the processes to progress as long as the input data of this provided action is available. In Case 5, when P_1 is in state 2 and P_2 in 2′, c_2 is required but P_1 cannot perform its matching action $\overline{c_1}$ at this state, so we add the

dual action to partial trace so as to allow P_1 to progress and reach state 3. In state 3, P_1 can synchronise with P_2 using the matching $c_1 \mapsto \overline{c_2}$.

These simple cases illustrate the gist of the recursive algorithm we devise to synthesise the mediator (see Algorithm 2). The algorithm starts by checking the basic configuration where both processes reach their final states and where the mediator is the END process (Lines 1−3). Then it considers the states of both processes and

Algorithm 2 : Mediator Synthesis

Require: P_1, P_2
Ensure: A mediator M
1: **if** $P_1 = \mathsf{END}$ and $P_2 = \mathsf{END}$ **then**
2: **return** END
3: **end if**
4: $M \leftarrow \mathsf{END}$
5: **for all** $P_i \xrightarrow{a} P'_{i=1,2}$ **do**
6: $matchingList \leftarrow$FindEligibleMatchings(a, P_i, P_{3-i})
7: **while** $\neg found$ and $matchingList \neq \emptyset$ **do**
8: $Match(X_1, X_2) \leftarrow$ selectMatching$(matchingList)$
such that $P_i \overset{X_1}{\Rightarrow} P''_i$ and $P_{3-i} \overset{X_2}{\Rightarrow} P'_{3-i}$
9: $M' \leftarrow$ SynthesiseMediator(P''_i, P'_{3-i})
10: **if** $M' \neq$ **fail then**
11: $found \leftarrow true$
12: $M_{m-n}(X_1, X_2) \leftarrow$GenerateMatchingProcess$(X_1, X_2)$
13: $M'' \leftarrow M_{m-n}(X_1, X_2); M'$
14: **end if**
15: **end while**
16: **if** $\neg found$ and $\exists \overline{\beta} \mid P_{3-i} \xrightarrow{\overline{\beta}} P'_{3-i}$ **then**
17: $M' \leftarrow$SynthesiseMediator(P_i, P'_{3-i})
18: **if** $M' \neq$ **fail then**
19: $found \leftarrow true$
20: $M_{\overline{\beta}} \leftarrow$ GenerateExtraProvidedActionProcess$(\overline{\beta})$
21: $M'' \leftarrow M_{\overline{\beta}}\ ; M'$
22: **end if**
23: **end if**
24: **if** $\neg found$ **then**
25: **return** fail
26: **end if**
27: $M \leftarrow M|M''$
28: **end for**
29: **return** M

for each enabled required action a, it calculates the list of matchings that can be applied, i.e., pairs (X_1, X_2) such that X_1 starts with a and P_2 is ready to engage in X_2 (Line 6). It selects one of them and makes a recursive call to test whether it can lead to a valid mediator (Lines 8−9). If that is the case, it generates the matching process associated with the selected matching and puts it in sequence with the returned mediator. Otherwise, it tries another matching until a valid matching is found or all the possible matchings have been tested (Lines 7−15). In the latter case, it checks whether the mediator can bypass a provided action in order to obtain a valid mediator, which corresponds to Case 5 in Figure 4.7. In this situation, it generates the appropriate process $M_{\overline{\beta}}$ and puts it in sequence with the generated mediator (Lines 16−23). If the required action does not match with any action given the states of both processes, the algorithm fails (Lines 24−26). Otherwise, it adds the new trace to the previously calculated mediator (Line 27). The algorithm explores all the outgoing transitions labelled with required actions (Lines 5−28) in order to make sure that whatever execution components perform, it will not lead to a deadlock. we do not systematically explore transitions labelled with provided actions because the synchronisation is triggered by required actions, which initialise the interaction by sending the input data.

The mediator M hence synthesised guarantees that P_1 and P_2 interact correctly, i.e., that the parallel composition $P_1 \| M \| P_2$ is deadlock free.

Theorem 1. *if $P_1 \leftrightarrow_M P_2$ then the parallel composition $P_1 \| M \| P_2$ is deadlock free.*

Proof. By construction, the mediator is synthesised only if both P_1 and P_2 reach an END state. In addition, at any given state s of any of P_1 or P_2, any transition associated with a required action α such that $s \stackrel{\alpha}{\Rightarrow} s'$ is involved in some matching $\alpha \mapsto \overline{\beta}$ and hence is associated with some matching process $M_{m-n}(\alpha, \overline{\beta})$ that is ready to engage in the sequence of dual provided actions, i.e., $M_{m-n}(\alpha, \overline{\beta})$ is in state s_m such that $s_m \stackrel{\overline{\alpha}}{\Rightarrow} s'_m$. Any transition associated with a provided action $\overline{\beta}$ such that $s \stackrel{\overline{\beta}}{\Rightarrow} s'$: (i) synchronises with a matching process $M_{m-n}(\alpha, \overline{\beta})$ if there exists a matching $\alpha \mapsto \overline{\beta}$ involving it, (ii) is an extra provided action, in which case it is associated with $s \stackrel{\beta}{\Rightarrow} s'$ from the $M_{\overline{\beta}}$ process, or (iii) is never triggered. □

Example. Figure 4.8 depicts an extract of the LTSs representing the behaviour of *WDAV* and *GDocs*. To simplify the presentation, the actions are represented using just the operation concept. As a result of the interface matching computation, Lock and Unlock match with SetSharingProperties, and MoveFile matches with DownloadDocument, UploadDocument, and DeleteDocument while the last two actions can be executed in any order. When both processes are at their initial states (1 and $1'$ respectively), the only applicable matching is Lock $\mapsto$ SetSharingProperties since *WDAV* is only able to perform this action. After applying this matching, *WDAV* goes to state 2, *GDocs* remains in state $1'$, and a partial trace of the mediator is created from $11 \rightarrow 11a \rightarrow 21a$. This is similar to Case 4 described in Figure 4.7. Then, *WDAV* can loop on the MoveFile required action, one of the possible matchings is chosen since both are applicable as *GDocs* loops on the three provided actions, DownloadDocument, UploadDocument, and DeleteDocument. *WDAV* stays in state 2, *GDocs* also remains in $1'$ while the mediator is augmented with the trace $21a \rightarrow 21b \rightarrow 21c \rightarrow 21d \rightarrow 21a$.

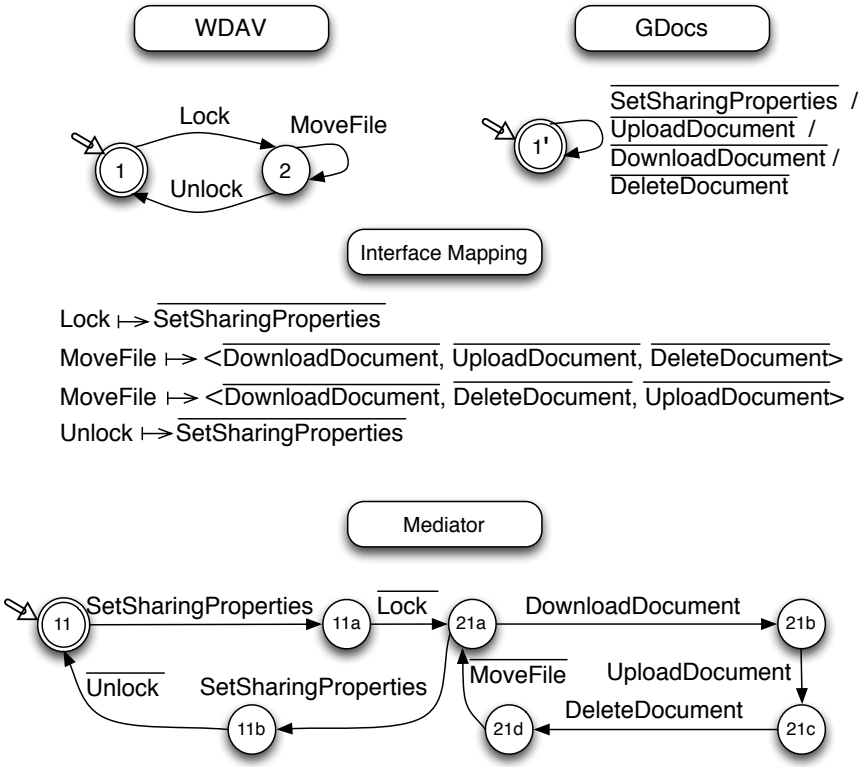

Figure 4.8. Illustrating the synthesis of a mediator between *WDAV* and *GDocs*

This is similar to Case 3 represented in Figure 4.7. *WDAV* can also branch on Unlock , which matches with SetSharingProperties and results in the trace $21a \rightarrow 11b \rightarrow 11$ in the mediator. This is again similar to Case 4 described in Figure 4.7. Finally, both processes reach their final states and the mediator is successfully created.

4.5 Summary

In this chapter, we presented an automated approach for the synthesis of mediators that guarantee the correct interaction between functionally-compatible components at the application layer. The synthesis of mediators is performed in two steps. First, we infer the semantic correspondence between the actions required by one component and those provided by the other by using constraint programming and ontology reasoning. Therefore, we incorporated the use of ontology reasoning within constraint solvers by defining an encoding of the ontology relations using arithmetic operators supported by the solvers. Then, we analyse the behaviours of components so as to generate the mediator which combines the mapping processes in a way that guarantees that the two components progress and reach their final states without errors. Nevertheless, only when mediators include all the details about the interaction of components, can interoperability be achieved. Hence, the next chapter describes how the synthesised mediators are refined in order to include the middleware details.

Chapter 5

From Abstract to Concrete Mediators

"The artist makes things concrete and gives them individuality."

— Paul Cézanne, painter (1839-1906)

Finding the right level of abstraction to achieve interoperability is crucial. On the one hand, reasoning about the meaning of the actions used by the components and analysing components' behaviours formally is essential to synthesise correct mediators. On the other hand, only when mediators include all the details about the interaction of components, can interoperability be achieved. The concretisation of mediators bridges the gap between the application level, which provides the abstraction necessary to reason about interoperability and synthesise mediators, and the middleware-level, which provides the techniques necessary to implement these mediators. Concretisation entails the instantiation of the data structures expected by each component and their delivery according to the interaction pattern defined by the middleware, based on which the component is implemented. We first consider the concretisation of mediators in the case of components implemented using the same middleware. As an illustration, we use the file management example introduced in the previous chapter, where both components, the WebDAV client and Google Docs service, are based on HTTP. We move to the concretisation of mediators between components implemented using different middleware solutions, which are based on the same interaction pattern. We illustrate the approach using the weather example introduced in Chapter 2, where *C2* is a Web Service client and *Weather Station* is a CORBA service. Finally, we present the most general case of components implemented using middleware that are based on different interaction patterns. As an

illustration, we use an example from the GMES case study which consists of two positioning components based on RPC and publish/subscribe respectively.

5.1 The Case of the Same Middleware

When both components are implemented using the same middleware, the mediator is only responsible for reconciling the differences in the components' interfaces and behaviours at the application layer. As no further mediation is necessary at the middleware layer, it suffices for the mediator to transform the required and provided actions used at the application layer into network messages, which is performed using the appropriate parsers and composers, as explained in Chapter 3. During the automated synthesis of mediators, presented in Chapter 4, we focused on the semantics of the actions of the components' interfaces to generate matching processes, which specify how sequences of actions required by one component can be achieved using sequences of actions provided by the other component. Reconciling the differences in the components' interfaces implies making explicit how the input/output data of these actions are actually translated in order to provide each component with the data it expects at the right moment and in the right format. In other words, the matching processes have to be made more concrete by incorporating the data translations necessary to solve syntactic differences between the input and output data included in the actions required/provided by the components.

To give basic understanding of the necessary data translation, let us first consider a one-to-one matching process $M_{1-1}\left(\alpha, \overline{\beta}\right)$ between an action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ required by one component and an action $\overline{\beta} = \left\langle \overline{b}, I_b, O_b \right\rangle \in \mathcal{I}_2$ provided by the other component. $M_{1-1}\left(\alpha, \overline{\beta}\right)$ must translate the input data produced by the required action into the input data expected by the provided action. While the subsumption relation between the ontological concepts associated with the input data guarantees

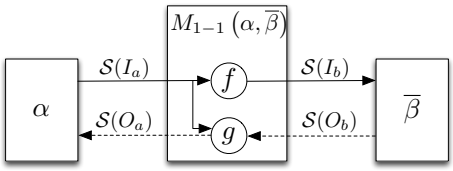

Figure 5.1. Illustrating data translation for one-to-one matching

that this translation is possible, since I_b subsumes I_a, the implementation of this translation must consider the syntax of the data. We recall that the XML schema that defines the syntax of the input/output data represented by the ontology concept C as $\mathcal{S}(C)$. The concretisation of $M_{1-1}\left(\alpha, \overline{\beta}\right)$ involves calculating a function f such that $\mathcal{S}(I_b) = f(\mathcal{S}(I_a))$. Likewise, the output data returned by the provided action must be translated into the output data expected by the required action. In addition, the input data previously produced by α can also be used, resulting in a function g such that $\mathcal{S}(O_a) = g\left(\mathcal{S}(I_a), \mathcal{S}(O_b)\right)$, which is defined since $I_a \sqcup O_b \sqsubseteq O_a$. Figure 5.1 illustrates the use of the f and g translation functions in the concretisation of the $M_{1-1}\left(\alpha, \overline{\beta}\right)$ matching process.

Let us now consider a many-to-many matching process $M_{m-n}(X_1, X_2)$ between a sequence of actions $X_1 = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ such that $O_{a_{i=1..l-1}} = \emptyset$, $1 \leq l \leq m$ required by one component and a sequence of actions

$X_2 = \left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{j=1..n}$ provided by the other component.

In this case, we postulate the existence of functions $f_{i=1..n}$ and $g_{j=l..m}$ such that:

1. $\mathcal{S}(I_{b_1}) = f_1\left(\mathcal{S}(I_{a_1}), \ldots, \mathcal{S}(I_{a_l})\right)$,
2. $\mathcal{S}(I_{b_i}) = f_i\left(\mathcal{S}(I_{a_1}), \ldots, \mathcal{S}(I_{a_l}), \mathcal{S}(O_{b_1}), \ldots, \mathcal{S}(O_{b_{i-1}})\right)$ for $i = 2..n$, and
3. $\mathcal{S}(O_{a_j}) = g_j\left(\mathcal{S}(I_{a_1}), \ldots, \mathcal{S}(I_{a_j}), \mathcal{S}(O_{b_1}), \ldots, \mathcal{S}(O_{b_n})\right)$ for $j = l..m$.

The translation function f_1 indicates that the mediator uses the data produced by the l first required actions to compute the input data necessary to execute the first provided action $\overline{\beta_1}$. Each translation function $f_{i=2..n}$ serves to calculate the input data for executing the provided action $\overline{\beta_i}$ based on the data produced by the l first required actions together with the output data resulting from the execution of the preceding $i-1$ provided actions. Once the n provided actions have been executed, the output data for the remaining required actions can be produced. Finally, each translation function $g_{j=l..m}$ computes the output data for the j^{th} required action based on the cumulated input data of the j required actions together with the output data of the n provided actions.

Consider, for example, the matching process between the <MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}> action required by the WebDAV client and the sequence of provided actions <<$\overline{\text{DownloadDocument}}$,

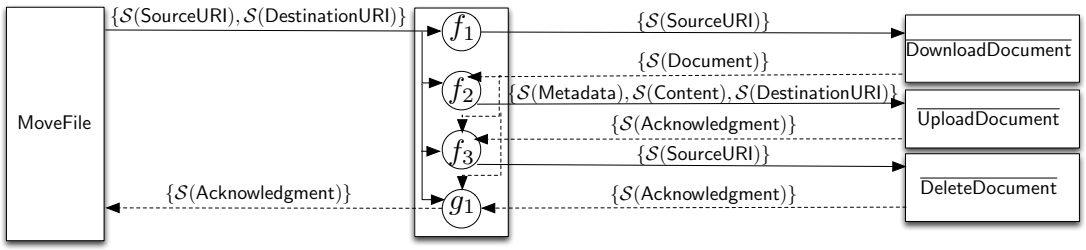

Figure 5.2. Concretising M_{1-n}(ReadFile, <$\overline{\text{DownloadDocument}}$, $\overline{\text{UploadDocument}}$, $\overline{\text{DeleteDocument}}$>)

{SourceURI}, {Document}>, <$\overline{\text{UploadDocument}}$, {Metadata, Content, DestinationURI},{Acknowledgment}>, <$\overline{\text{DeleteDocument}}$, {SourceURI}, {Acknowledgment}>> defined by the Google Docs service. As depicted in Figure 5.2, the concrete matching process must have a translation function f_1 that produces the input data $\mathcal{S}$(SourceURI), for $\overline{\text{DownloadDocument}}$ to execute. The translation function f_2 must compute the input data required by $\overline{\text{UploadDocument}}$ based on the input data of MoveFile together with the Document returned by $\overline{\text{DownloafDocument}}$. Likewise, f_3 must calculate the input data of $\overline{\text{DeleteDocument}}$ based on the input data of MoveFile together with the output data returned by both $\overline{\text{DownloafDocument}}$ and $\overline{\text{UploadDocument}}$. Finally, g_1 computes the output data, $\mathcal{S}$(Acknowledgment), necessary for the MoveFile action.

5.1.1 From Ontological Relations to Data Translation Functions

To compute the translation functions $f_{i=1..n}$ and $g_{j=l..m}$, in addition to the domain ontology, we also use XML schema matching techniques to identify related elements between the schema of the data given as input to the translation function and the schema of the data that should be returned as output. The use of ontologies together with XML schema matching techniques is motivated by several observations. First, it is often tedious to annotate the simple types or the attributes of the XML schema associated with the input/output data. Rather, annotating complex types is both more common and more relevant. For example, it is both more common and useful to annotate Metadata instead of individual attributes such as Title or CreationDate. Second, even though there are many off-the-shelf schema-matching

tools, they are often able to detect only one-to-one correspondences between the attributes of the schema automatically and require user intervention for more complex correspondences [SE05]. The reason is that schema-matching tools often compute the correspondences by estimating the similarity of attribute names or types, while more knowledge of the domain is necessary to calculate complex correspondences. Ontologies are more suitable for expressing complex correspondences. Therefore, we use the ontology relations that we used during the generation of matching processes to compute complex correspondences, as explained in the following.

Subsumption. We are given two data concepts D_s and D_t such that $D_s \sqsubseteq D_t$. Each data concept is associated with an XML schema, $\mathcal{S}(D_s)$ and $\mathcal{S}(D_t)$ respectively, which defines the concrete structure of the data. The aim is to find a translation function f that specifies how the XML elements of $\mathcal{S}(D_t)$ are computed using the XML elements of $\mathcal{S}(D_s)$, i.e., $\mathcal{S}(D_t) = f(\mathcal{S}(D_s))$.

In the case where both $\mathcal{S}(D_s)$ and $\mathcal{S}(D_t)$ are simple types, they are either described using the same built-in XML data type (e.g., string, integer, boolean, float, decimal), in which case the translation function f is a simple assignment; or a conversion function is necessary to transform the data type of the source schema into that of the target schema. In the latter case, we use pre-defined conversion functions to perform this task. Specifically, as we implemented the concretisation using Java, we use the facilities it provides to perform data type conversions.

In the case where $\mathcal{S}(D_s)$ or $\mathcal{S}(D_t)$ include complex types, we use a schema matching tool, Harmony [SMH+10], to identify correspondences between the elements of $\mathcal{S}(D_s)$ and $\mathcal{S}(D_t)$. Harmony is a schema-matching tool that combines multiple algorithms to evaluate correspondences between a source and a target schema. Each algorithm identifies correspondences using a different strategy. For example, one strategy consists in counting the words appearing in the definitions or documentation of XML elements. Another strategy consists in comparing the names of the XML elements using a thesaurus. For each pair of elements from the source and target schema, an algorithm establishes a confidence score ranging from -1 to +1 where -1 indicates that there is definitely no correspondence, +1 indicates a definite correspondence, and 0 that there is not enough evidence to make any assertion about the relation between the two elements. Then, the confidence scores of the individual algorithms are combined into a single confidence score according to two criteria.

First, the weighted sum of the confidence scores of all the algorithms is calculated. The weights can be customised but by default they are all given the same weight (for k algorithms, the weight is $1\backslash k$). Then, the confidence score is adjusted based on structural information: positive confidence scores propagate up to parent elements, and negative confidence scores trickle down to sub-elements. Intuitively, two elements are unlikely to match if their parent elements do not match. Once all the confidence scores have been identified, for each element $e_{i=1...n}$ in the target schema, we select a corresponding element $e'_{i=1...n}$ in the source schema with the highest confidence score. We calculate the function f_i to transform $e_{i=1...n}$ into $e'_{i=1...n}$, which is either an assignment if they are of the same data type or a data type conversion. The translation function f is then defined as the composition of individual functions, i.e., $f = f_1 \circ f_2 \circ \cdots \circ f_n$.

Disjunction. We are given data concepts $D_{s_{i=1..n}}$ and D_t such that $D_t = \bigsqcup_{i=1}^{n} D_{s_i}$. Disjunction indicates that every instance of $D_{s_{i=1..n}}$ is also an instance D_t. The aim is to compute the translation function f that specifies how the XML elements of $\mathcal{S}(D_t)$ are computed using the XML elements of source schemas, i.e., $\mathcal{S}(D_t) = f\left(\mathcal{S}(D_{s_1}), \ldots, \mathcal{S}(D_{s_n})\right)$.

First, we compute for each $D_{s_{i=1..n}}$ a translation function $\mathcal{S}(D_t) = f_i(\mathcal{S}(D_{s_i}))$. We use the same solution as subsumption: we distinguish between the case of simple types and that of complex types and use conversion functions between different data types. Then, we build a function *select* that chooses the translation functions f_i to execute according to the data received. Specifically, we use the `instanceof` operator of Java to implement the *select* function. Hence, the translation function f is defined as $f = select\left(f_1, \ldots, f_n\right)$.

Aggregation. We are given data concepts $D_{s_{i=1..n}}$ and D_t such that $D_t = \bigoplus_{i=1}^{n} D_{s_i}$. Aggregation indicates that every instance of D_t includes an instance of every $D_{s_{i=1..n}}$. We have to find the translation function f that specifies how the XML elements of $\mathcal{S}(D_t)$ are computed using the XML elements of source schemas, i.e., $\mathcal{S}(D_t) = f\left(\mathcal{S}(D_{s_1}), \ldots, \mathcal{S}(D_{s_n})\right)$.

We start by creating the XML schema that concatenates all the schemas, that is, $S_{agg} = concat\left((\mathcal{S}(D_{s_1}), \ldots, \mathcal{S}(D_{s_n})\right)$. Then, as for subsumption, we rely on XML

schema matching to define the function f_{agg} that specifies how the XML elements of $\mathcal{S}(D_t)$ are computed using the XML elements of S_{agg}. Hence, the translation function is the composition of the two functions, i.e., $f = f_{agg} \circ concat$.

5.1.2 Application to the File Management Example

In the previous chapter, we used the file management example to illustrate the automated synthesis of mediators at the application layer. The mediator enables a WebDAV client (*WDAV*) to interact successfully with the Google Docs service (*GDocs*). As depicted in Figure 5.3, although both *WDAV* and *GDocs* are based on HTTP, the mediator has to further include translation functions to convert the data included in the HTTP messages sent by one component into that expected by the other component.

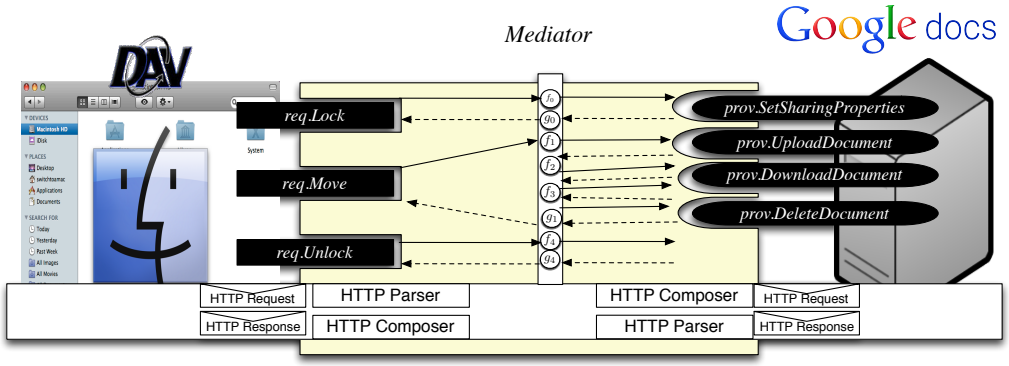

Figure 5.3. Illustrating concretisation in the file management example

Let us consider the case of the translation function between Document, used by *GDocs*, and File, used by *WDAV*, such that Document $\sqsubseteq$ File (see Figure 4.2 illustrating the file ontology), which is for example necessary for M_{1-1}(ReadFile, $\overline{\text{DownloadDocument}}$). The XML schema for Document is as follows:

```
<entry xmlns:gd="http://schemas.google.com/g/2005"
       gd:etag="'HhJSFgpeRyt7ImBq'">
  <ref>https://docs.google.com/feeds/id/pdf%3AtestPdf</ref>
  <published>2012-04-09T18:23:09.035Z</published>
  <updated>2012-04-09T18:273:09.035Z</updated>
  <app:edited xmlns:app="http://www.w3.org/2007/app">
    2009-06-18T22:16:02.388Z
  </app:edited>
```

```
    <title>PDF's Title</title>
    <content type="application/pdf"
      src="https://doc-04-20-docs.googleusercontent.com/docs/
        secure/m71240...U1?h=1630126&amp;e=download&amp;gd=true"/>
    <link rel="alternate" type="text/html"
      href="https://docs.google.com/ fileview?
      id=testPdf&amp;hl=en"/>
    <author>
      <name>benamel</name>
      <email>benamel@gmail.com</email>
    </author>
    <gd:resourceId>pdf:testPdf</gd:resourceId>
    <gd:lastViewed>2012-06-18T22:16:02.384Z
    </gd:lastViewed>
    <gd:quotaBytesUsed>108538</gd:quotaBytesUsed>
    <docs:writersCanInvite value="false"/>
    <docs:md5Checksum>2b01142f7481c7b056c4b410d28f33cf
    </docs:md5Checksum>
</entry>
```

while the associated XML schema for File is the following:

```
<w:response>
      <w:href>https://docs.google.com/feeds/id/pdf%3AtestPdf</w:href>
      <w:propstat>
        <w:status>HTTP/1.1 200 OK</w:status>
        <w:prop>
          <w:displayname>testPdf.pdf</w:displayname>
          <w:author>benamel</w:author>
        </w:prop>
      </w:propstat>
</w:response>
```

The bold tags represent the assignments that are performed by the translation function. For example, the value of `ref` defined by *GDocs* is assigned to `href` used by WebDAV. Likewise, the value of the `name` tag, within the `author` complex type, defined by *GDocs* is assigned to `author` used by *WDAV*. Note though that the `status` tag does not have an equivalent, but it supports a default value, which we assign to it.

The translation functions are internal to the mediator and cannot be perceived by an external observer. Let us now consider the observable behaviour of the whole

system. Using HTTP to implement component interactions means that each required action is realised by sending an HTTP request and receiving an HTTP response while a provided action is realised by receiving an HTTP request and sending the corresponding HTTP response. This interaction is specified as follows:

$$
\begin{aligned}
&HTTPClient\ (X =' op) = (req.[X] \rightarrow sendHTTPRequest[X] \\
&\qquad \rightarrow receiveHTTPResponse[X] \rightarrow HTTPClient). \\
&HTTPServer\ (X =' op) = (prov.[X] \rightarrow receiveHTTPRequest[X] \\
&\qquad \rightarrow sendHTTPResponse[X] \rightarrow HTTPServer). \\
&HTTPGlue\ (X =' op) \quad = (sendHTTPRequest[X] \rightarrow receiveHTTPRequest[X] \\
&\qquad \rightarrow sendHTTPResponse[X] \rightarrow receiveHTTPResponse[X] \\
&\qquad \rightarrow HTTPGlue). \\
&\|HTTPImplementation = (\ HTTPClient \parallel HTTPServer \parallel HTTPGlue).
\end{aligned}
$$

To facilitate the presentation, we represent each action using only its name. We consider part of the behaviour of *WDAV*, which involves locking a file, moving it and unlocking it again, which is specified as follows:

$$
\begin{aligned}
&WDAV = (req.lock \rightarrow P), \\
&P \quad\ = (\ req.moveFile \rightarrow P \mid req.unlock \rightarrow WDAV).
\end{aligned}
$$

The actions provided by *GDocs* can be executed in any order:

$$
\begin{aligned}
GDocs = (&prov.setSharingProperties \rightarrow GDocs \\
\mid\ &prov.uploadDocument \rightarrow GDocs \\
\mid\ &prov.downloadDocument \rightarrow GDocs \\
\mid\ &prov.deleteDocument \rightarrow GDocs).
\end{aligned}
$$

The mediator coordinates the behaviours of *WDAV* and *GDocs* as follows:

$$
\begin{aligned}
&Mediator = (g.req.setSharingProperties \rightarrow w.prov.lock \rightarrow P2), \\
&P \quad\ = (\ g.req.downloadDocument \rightarrow g.req.uploadDocument \\
&\qquad \rightarrow g.req.deleteDocument \rightarrow w.prov.moveFile \rightarrow P2 \\
&\qquad \mid\ g.req.setSharingProperties \rightarrow w.prov.unlock \rightarrow Mediator).
\end{aligned}
$$

The behaviour of the resulting composed system is then the following:

$$
\begin{array}{lll}
\mathsf{set}\ WDAV\_operations & = & \{lock, moveFile, unlock\} \\
\mathsf{set}\ GDocs\_operations & = & \{setSharingProperties, uploadDocument, downloadDocument, \\
 & & \quad deleteDocument\} \\
\\
\|FileManagementSystem = & (& w : WDAV \\
 & \| & g : GDocs \\
 & \| & Mediator \\
 & \| & (\mathsf{forall}[op : WDAV\_operations]\ w : HTTPImplementation(\mathrm{op})) \\
 & \| & (\mathsf{forall}[op : GDocs\_operations]\ g : HTTPImplementation(\mathrm{op}))).
\end{array}
$$

We use the prefixes w and g to decouple the behaviours of $WDAV$ and $GDocs$ since they only interact through the mediator. We can easily check that $\|FileManagementSystem$ is free from deadlocks.

5.2 The Case of Different Middleware Based on the Same Interaction Pattern

Communication in distributed systems is always based on low-level message passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives proposed by middleware solutions [TVS06]. Middleware facilitates communication and coordination between components in distributed systems by defining [ICG07]: (i) an Interface Description Language (IDL) for specifying the interfaces of components and the associated operations, and data types, (ii) a discovery protocol to address and locate the components that are available in the environment, and (iii) an interaction protocol that coordinates the behaviour of different components and enables them to collaborate. While middleware solutions and implementations define diverse IDLs and message formats, their interaction protocols follow comparably few interaction patterns, a.k.a., communication paradigms/types [CDKB12] or coordination models/paradigms [ICG07]. An interaction pattern defines the rules to coordinate the behaviours of the components. We start by presenting the principles and characteristics of each interaction pattern. We also introduce, for each interaction pattern, an ontology that models the essential primitives of this interaction pattern, which we use to specify the behaviours expected by the components implemented using a middleware solution based on this interaction pattern as well as how these behaviours are coordinated. A specific mid-

dleware solution is modelled using the ontology (or ontologies) that represents the interaction pattern on which the middleware solution is based. Finally, we show how the ontology can be used to concretise mediators between components implemented using different middleware solutions based on the same interaction pattern and use the weather example from Chapter 2 to illustrate the approach.

5.2.1 Ontology-based Modelling of Middleware Interaction Patterns

5.2.1.1 Remote Procedure Call

The concept of a remote procedure call (RPC) [BN84] represents the most common interaction pattern in distributed systems. This approach directly and elegantly supports client/server interactions with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally. The interaction is supported by a pairwise exchange of messages from the client to the server and then from the server back to the client, with the first message containing the operation to be executed at the server and associated arguments and the second message containing any result of the operation. To interact according to RPC, the client and the server must agree on the format of the messages they exchange as well as the encoding of the data, which represent the arguments and results, enclosed in these messages.

An RPC-based middleware hides the encoding and decoding of arguments and results as well as the passing of messages using communication modules, *stubs*, that permit the client and server to use the operations as if they were local. RPC-based middleware solutions are often associated with libraries to generate, either at compile time or runtime, the client and server stubs based on the interface definition. The strict request-reply message exchange is unnecessary when there is no result to return. RPC middleware solutions may also provide facilities for what are called asynchronous RPCs, by which a client immediately continues its execution after issuing the RPC request.

Figure 5.4 depicts the RPC ontology that makes explicit the main concepts underpinning RPC middleware solutions. The ontology defines the primitives that support message exchange. On the client side, the invocation of an operation is achieved using sendRequest, which specifies the operation invoked using methodName and the

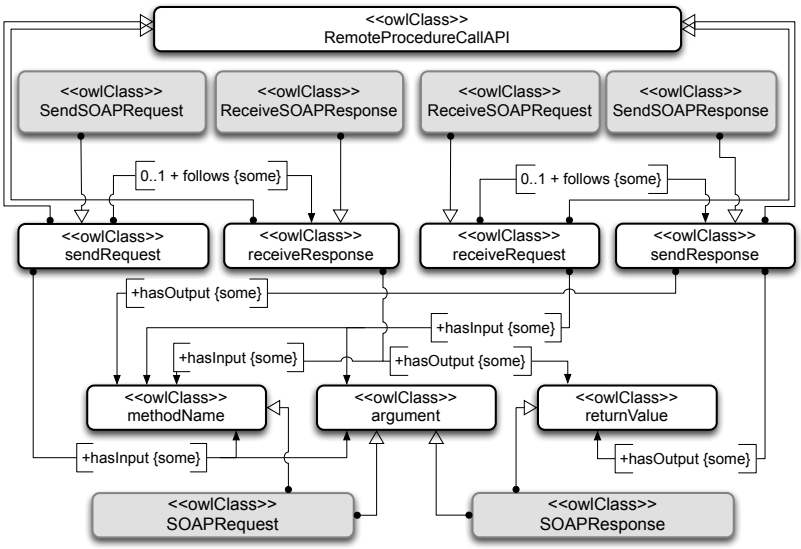

Figure 5.4. The RPC ontology with SOAP specialisation

associated argument data argument, possibly followed by a receiveResponse, which includes the operation invoked methodName together with the results returnValue. The server gets the operation call using the receiveRequest primitive, which includes methodName and argument, if the result of this operation is not empty, the server returns it using the sendResponse primitive. Figure 5.4 further shows an example of the use of SOAP request/response middleware to implement RPC. Even though SOAP supports the sending and reception of messages independently, it is often used to realise RPC-based interactions, especially in the context of Web Services.

The client and server behaviours are then coordinated as follows:

$$
\begin{aligned}
&Client\ (X =' op) && = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client).\\
&Server\ (X =' op) && = (receiveRequest[X] \rightarrow sendResponse[X] \rightarrow Server).\\
&RPCGlue\ (X =' op) && = (sendRequest[X] \rightarrow receiveRequest[X]\\
& && \quad \rightarrow sendResponse[X] \rightarrow receiveResponse[X] \rightarrow RPCGlue).\\
&\|RPCInteraction && = (\ (\mathsf{forall}[op : Interface]Client(op))\\
& && \quad \| \quad (\mathsf{forall}[op : Interface]RPCGlue(op))\\
& && \quad \| \quad (\mathsf{forall}[op : Interface]Server(op))).
\end{aligned}
$$

where *op* defines the operation signature that is made up of the methodName, argument, and returnValue. The precise definition of the $Interface$ set is specific to the application implemented on top of the middleware.

RPC represents an interaction pattern that implies a direct relationship between components explicitly sending and receiving messages, where the client initialises the interaction by sending the request. In contrast, a number of techniques have emerged whereby interaction is indirect, through a third entity, allowing a higher degree of decoupling between components. In particular, components may not need to know the destination of their messages, i.e., space decoupling, and they also do not need to exist at the same time, i.e., time decoupling.

5.2.1.2 Distributed Shared Memory

While RPC allows developers to invoke operations as if they were available locally, Distributed Shared Memory (DSM) provides developers with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces. DSM is primarily intended for parallel applications but is also used for any distributed application in which individual shared data items can be accessed directly. DSM is in general less appropriate for client/server interactions, where clients usually access server-held resources using an explicit interface (for reasons of modularity and protection). Still, servers can provide DSM that is shared between clients.

A DSM-based middleware enables components to read and write data in the shared memory, regardless of the exact location of the data. Nevertheless, the structure of the shared data is defined at the application layer and the middleware does not provide any guarantee about when data is made available and how long it will reside in the shared memory. In other words, the synchronisation between the readers and writers also needs to be managed at the application layer.

Figure 5.5 illustrates the DSM ontology we create to represent the main concepts of this interaction pattern. Two primitives are used: write, which adds data to the shared memory and read, which retrieves data from the shared memory. The dataChannel concept defines how to select the data to read. Figure 5.5 also shows how the ontology can be used to specify LIME [MPR06]. LIME is a tuple space middleware intended for mobile applications. Data is structured as tuples, which are sequences of fields, each of which has a specific data type. The write primitive can take two forms: out for writing a single tuple or outg to write a set of tuples. Likewise, the read primitive can also take many forms: it can delete or not the tuple once read (in or rd), block until a tuple is available (inp and rdp), and read a set of tuples (ing and rdg). In order to select the tuples to read, a tupleTemplate must be specified.

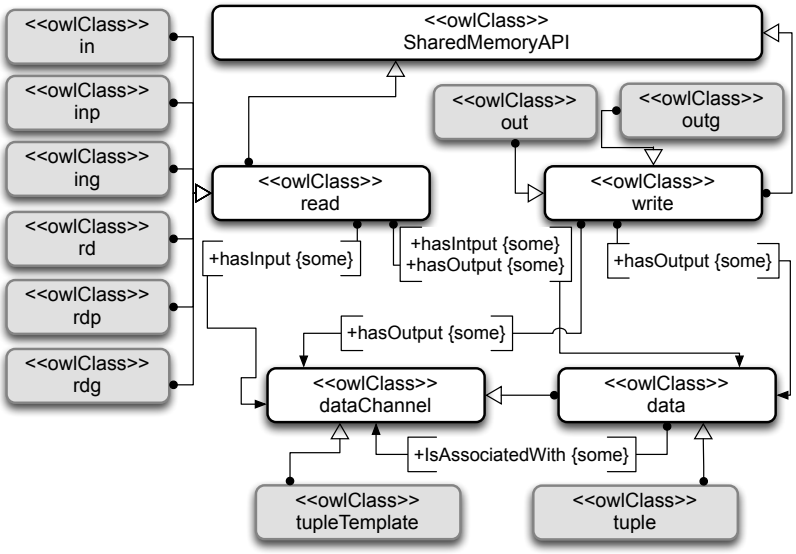

Figure 5.5. The DSM ontology specialised with LIME

tupleTemplate defines a filter over the tuples by specifying either a value that a field must have, or the type of the field.

The coordination of the behaviours of components, which can be considered as readers or writers, is achieved through the shared memory as follows:

$$
\begin{array}{ll}
Writer(X =' data) & = (write[X] \rightarrow Writer). \\
Reader(X =' data, Y =' dataChannel) & = (read[X][Y] \rightarrow Reader). \\
SharedMemory(X =' data) & = (write[X] \rightarrow P[X]), \\
P[X][a : DataChannels] & = (\mathsf{if}\ (X\ matches\ a)\ \mathsf{then}\ \ read[X][a : DataChannels] \\
 & \quad \rightarrow P[X]). \\
\| DSMInteraction & = (\ (\mathsf{forall}[data : Data]\, Writer(data)) \\
 & \quad \|\ \ (\mathsf{forall}[data : Data]\, SharedMemory(data)) \\
 & \quad \|\ \ (\mathsf{forall}[data : Data][dataChannel : DataChannels] \\
 & \qquad Reader(data, dataChannel))).
\end{array}
$$

As FSP supports only finite state models, we must represent *data* and *dataChannel* as sets. The precise definition of these sets depends on the application that uses the DSM. Note that there is one process P per data item, which deals with the several reads assuming that the data are persistent, i.e., the data can be read infinitely often. The *matches* function indicates whether the data channel specified in the read corresponds to the data managed by P. It is the role of the middleware to implement

the *matches* function. Note that in DSM, the writer initialises the interaction by making the data available on the shared memory.

5.2.1.3 Publish/Subscribe

Many applications require the dissemination of information or items of interest from a large number of producers to a similarly large number of consumers. Publish/subscribe middleware solutions provide an intermediary service, a *broker*, that efficiently ensures that information generated by producers is delivered to the consumers that want to receive it. In other words, publish/subscribe middleware solutions (sometimes also called distributed event-based middleware) allow subscribers to register their interest in an event, or a pattern of events, and ensure that they are asynchronously notified of events generated by publishers. The task of the publish/subscribe middleware is to match subscriptions against published events and ensure the correct delivery of event notifications. A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many interaction pattern.

The expressiveness of publish/subscribe middleware solutions is determined by the type of event subscriptions they support: either subscriptions are made using specific topics (also referred to as subjects) which the events belong to, or based on the content of the event.

Figure 5.6 depicts the event middleware ontology representing the main concepts

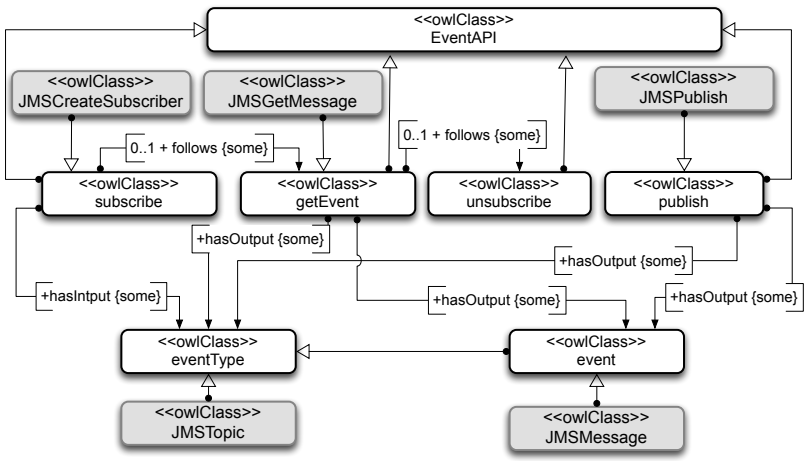

Figure 5.6. Event middleware ontology specialised with JMS

related to the publish/subscribe interaction pattern. A publisher disseminates an event event using the publish primitive and subscribers express an interest in a set of events using the subscribe primitive, which is parameterised by eventType that defines a filter over the set of all possible events. The events are delivered to subscribers using the getEvent primitive. Subscribers can later revoke this interest using the unsubscribe primitive. Figure 5.6 also illustrates how this ontology can be used to specify JMS. An event is represented as JMSMessage and belongs to a specific JMSTopic. Events are published using the JMSPublish primitive. Subscribers register to topics using JMSCreateSubscriber, and receive notifications using a callback method, JMSGetMessage.

Coordinating the behaviours of publisher and subscriber is achieved using a broker as follows:

$$
\begin{array}{ll}
Publisher(X =' event) & = (publish[X] \rightarrow Publisher). \\
Subscriber(X =' event, Y =' eventType) & = (subscribe[Y] \rightarrow getEvent[Y] \rightarrow Subscriber). \\
Broker & = P, \\
P & = (\ subscribe[eventType : EventTypes] \\
 & \quad \rightarrow MATCH[eventType] \\
 & \quad \mid publish[Events] \rightarrow P), \\
MATCH[eventType : EventTypes] & = (\ publish[event : Events] \rightarrow \\
 & \quad \textsf{if}\ (event\ matches\ eventType)\ \textsf{then} \\
 & \qquad getEvent[event] \rightarrow MATCH[eventType] \\
 & \quad \textsf{else}\ \ MATCH[eventType]). \\
 & \\
\|PubSubInteraction & = (\ (\textsf{forall}[event : Events] Publisher(event)) \\
 & \quad \| \quad (Events : Broker)\{publish/Events.publish\} \\
 & \quad \| \quad (\textsf{forall}[event : Events][eventType : EventTypes] \\
 & \qquad Subscriber(event, eventType))).
\end{array}
$$

Similarly to DSM, we represent *event* and *eventType* as sets while the precise definition of these sets depends on the application that uses the publish/subscribe middleware. Note that we define several $MATCH$ processes, each of which manages the subscriptions related to one specific event type. The *matches* function indicates whether the published event is of the type managed by the specific $MATCH$ process. The middleware is in charge of implementing this function. Note that the publisher initialises the interaction by pushing the event to the broker.

To sum up, there are different interaction patterns that define specific rules to coordinate the behaviours of components. While we present and formalise the inter-

actions patterns most commonly used in the development of middleware solutions, we are aware that some middleware are not represented, e.g., stream-based middleware solutions. The case of streaming solutions is to be explored in future work. We also stress that many modern middleware solutions tend to support a combination of interaction patterns. For example, CORBA offers distributed event services to complement support for remote procedure calls.

When components are implemented using middleware solutions that are based on the same interaction pattern, besides the translations and behavioural coordination at the application layer, the mediator is also responsible for message translations at the middleware layer. Therefore, the middleware ontologies serve as intermediary for the necessary translations. Assuming that appropriate parsers and composers, associated with specific middleware solutions, can be used to extract the necessary data from the network messages, as well as to create network messages given the necessary data, the synthesised mediator thus perceives a uniform interface offered by the middleware layer.

5.2.2 Application to the Weather Example

Going back to the weather example introduced in Chapter 2 where the goal is to achieve interoperability between, *C2*, which invokes a single SOAP operation to get weather information, and *Weather Station*, which is a CORBA server that provides temperature and humidity measures. The mediator must transform the $getWeather$ action required by *C2* into the sequence of actions $getTemperature$ and $getHumidity$ provided by *Weather Station*. The implementation of the mediator also requires dealing with differences between the SOAP requests/responses used by *C2* and the CORBA requests/responses used by *Weather Station*. Even though the format of the requests/responses is different, the interaction pattern is the same. Consequently, by using the appropriate parsers and composers, they can be transformed into primitives from the RPC ontology, as depicted in Figure 5.7.

We can further verify that using the software system composed of *C2*, *Weather Station*, and the synthesised mediator ($\|\, WeatherSystem$), as described in the following, is free from deadlocks.

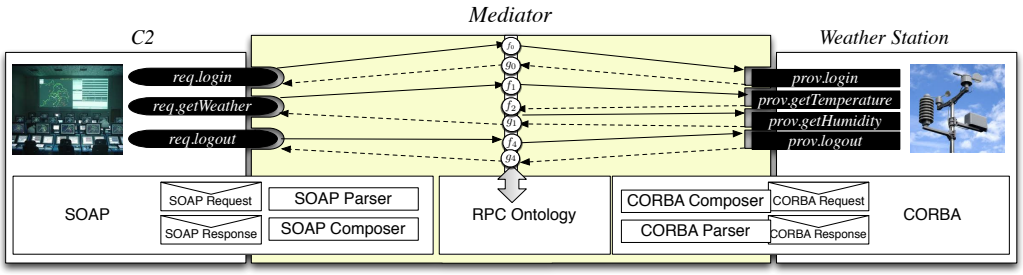

Figure 5.7. Illustrating concretisation in the weather example

set $C2\_weather\_actions = \{login, getWeather, logout\}$
set $WeatherStation\_actions = \{login, getTemperature, getHumidity, logout\}$

$$
\begin{array}{lcl}
C2\_weather\_role & = & (req.login \rightarrow P1), \\
P1 & = & (req.getWeather \rightarrow P1 \mid req.logout \rightarrow C2\_weather\_role). \\
 & & \\
WeatherStation\_role & = & (prov.login \rightarrow P2), \\
P2 & = & (\ prov.getTemperature \rightarrow P2 \\
 & \mid & \ \ prov.getHumidity \rightarrow P2 \\
 & \mid & \ \ prov.logout \rightarrow WeatherStation\_role). \\
 & & \\
Mediator & = & (c2.req.login \rightarrow c2.prov.login \rightarrow P), \\
P & = & (c2.req.getWeather \rightarrow ws.prov.getTemperature \\
 & & \rightarrow ws.sendCORBARequest.getHumidity \rightarrow P \\
 & \mid & c2.req.logout \rightarrow ws.prov.logout \rightarrow Mediator). \\
\parallel WeatherSystem & = & (\ c2 : C2\_weather\_role \\
 & \parallel & \ \ ws : WeatherService\_role \\
 & \parallel & \ \ Mediator \\
 & \parallel & \ \ (\mathsf{forall}[op : C2\_weather\_actions]\ c2 : SOAPImplementation(op)) \\
 & \parallel & \ \ (\mathsf{forall}[op : WeatherStation\_actions]\ ws : CORBAImplementation(op))).
\end{array}
$$

5.3 The Case of Middleware Based on Different Interaction Patterns

While middleware solutions are abound, there are comparably very few interaction patterns. Our approach to mediator concretisation seeks to identify, capture and separate the core of the middleware represented by the interaction pattern it uses from specific details related to the format of messages. We hypothesise that simple transformations between different interaction patterns can be elucidated through em-

pirical study. Therefore, we define a generic mapping of all interaction patterns into required/provided actions, and illustrate the transformations necessary to coordinate different interaction patterns.

5.3.1 Coordination across Interaction Patterns

Whether expressed as operation calls, data read and write, or event publication, component interactions mainly consists in the production and consumption of information. The production of information in the environment is modelled using provided actions while the consumption from the environment is modelled using required actions. All middleware solutions, regardless of the interaction pattern they are based on, provide an abstraction that represents required and provided actions. Figure 5.8 shows how the primitives associated with each interaction pattern, and defined in the associated ontology, are mapped to required/provided actions. In RPC, the server provides an action whose functionality is expressed by the $methodName$, it uses as input $argument$ and generates $returnValue$. The associated client requires this same action. In DSM, it is the writer that provides an action while the functionality is enclosed in the data itself as $data$ is associated with a specific $dataChannel$. The reader selects $data$ available on some $dataChannel$. In publish/subscribe, the publisher provides an action whose functionality is represented by the event type. The subscriber selectively consumes these events by subscribing to a specific $eventType$. Similarly to the DSM case, each event is associated with an event type.

Each matching process used during the generation of the mediator, associates a sequence of required actions from one component with a sequence of provided actions from the other component. Let us now present the different cases that need to be considered to concretise these matching processes.

RPC Client – DSM Writer. In this case, the mediator intercepts the request and converts the $methodName$ and $arguments$ into $dataChannel$. The mediator then uses $dataChannel$ to read $data$, which it transforms into the appropriate $returnValue$ and sends the response back to the client. This is formally specified as follows:

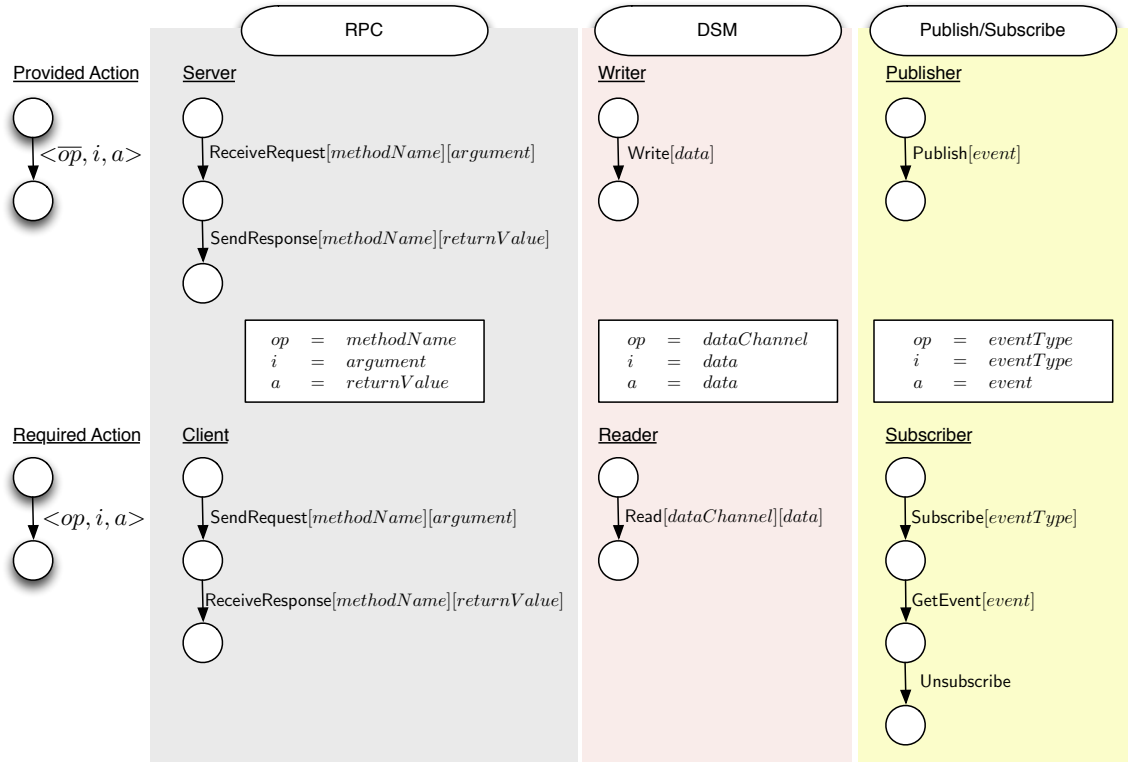

Figure 5.8. Mapping interaction patterns to required/provided actions

$$
\begin{aligned}
&Client\ (X =' op) \quad = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client).\\
&Writer(Y =' data) = (write[Y] \rightarrow Writer).\\
&RPC2DSMGlue(X =' op, Y =' dataChannel, Z =' data) = (receiveRequest[X] \rightarrow translate[X][Y]\\
&\qquad\qquad \rightarrow read[Y][Z] \rightarrow translate[Z][X] \rightarrow sendResponse[X]\\
&\qquad\qquad \rightarrow RPC2DSMGlue).\\
&\|RPC\text{-}DSM \qquad = (\ (\mathsf{forall}[op : Interface]\, Client(op))\\
&\qquad\qquad \|\quad (\mathsf{forall}[op : Interface]\, RPCGlue(op))\\
&\qquad\qquad \|\quad (\mathsf{forall}[data : Data]\, Writer(data))\\
&\qquad\qquad \|\quad (\mathsf{forall}[data : Data]\, SharedMemory(data))\\
&\qquad\qquad \|\quad (\mathsf{forall}[op : Interface][data : Data][dataChannel : DataChannels]\\
&\qquad\qquad\qquad RPC2DSMGlue(op, data, dataChannel))).
\end{aligned}
$$

where the sets $Interface$, $Data$, and $DataChannels$ are specific to the application, as well as the translations performed by the $RPC2DSMGlue$. We can easily verify that $\|RPC\text{-}DSM$ is free from deadlocks. Note though that we assume the availability of the shared memory.

RPC Client – Publisher. The mediator intercepts the request and converts the $methodName$ and $arguments$ into the appropriate $eventType$ to which it subscribes. Once the mediator receives a notification of a new event, it transforms it into $returnValue$ and sends the response to the client. This is specified as follows:

$$
\begin{aligned}
&Client\ (X =' op) && = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client).\\
&Publisher(X =' event) && = (publish[X] \rightarrow Publisher).\\
&RPC2PSGlue(X =' op, Y =' eventType, Z =' event) && = (receiveRequest[X] \rightarrow translate[X][Y]\\
& && \quad \rightarrow subscribe[Y] \rightarrow getEvent[Z] \rightarrow translate[Z][X]\\
& && \quad \rightarrow sendResponse[X] \rightarrow RPC2PSGlue).\\
&\|RPC\text{-}PS && = (\ (\mathsf{forall}[op : Interface]Client(op))\\
& && \quad \| \quad (\mathsf{forall}[op : Interface]RPCGlue(op))\\
& && \quad \| \quad (\mathsf{forall}[event : Events]Publisher(event))\\
& && \quad \| \quad (Events : Broker)\{publish/Events.publish\}\\
& && \quad \| \quad (\mathsf{forall}[op : Interface][eventType : EventType][event : Events]\\
& && \qquad RPC2PSGlue(op, eventType, event))).
\end{aligned}
$$

where the sets $Interface$, $Events$, and $Event$ are application-specific. However, $\|RPC\text{-}DSM$ may deadlock if the event was published before the subscription. To remedy this case, if the event middleware is topic-based, the mediator can anticipate the interaction and subscribe to the topic beforehand. Another possibility, although very inefficient, is to subscribe to all events and cache them. Finally, some middleware solutions (e.g., JMS) allow subscriptions to past events.

DSM Reader – RPC Server. This is a problematic case as neither the reader nor the server initiates the interaction. If the action provided by the server does not need any input data, the mediator may perform a polling and write the data returned in the shared memory. Otherwise, the mediator should be able to intercept the network message representing the read request, and invoke the server accordingly. Hence, the concretisation cannot be achieved using the primitives provided by the middleware solutions.

DSM Reader – Publisher. Unless operating at the network layer, the mediator cannot be concretised in this case. The reason is that in order to synchronise with the reader, the mediator must write the appropriate data in the shared memory. On the other hand, it can only get the output data to write if it subscribes to the appropriate

event type, which can only be performed by getting the read request. Fortunately, DSM-based middleware solutions (e.g., LIME and JavaSpaces) increasingly support publish/subscribe interactions as well.

Subscriber – RPC Server. As in the case of DSM Reader – RPC Server, both components are passive, meaning that neither of them initialises the interaction. Unless the mediator is able to intercept the network messages associated with subscriptions and then periodically poll the server, this situation cannot be handled.

Subscriber – DSM Writer. In this case as well, the mediator must be able to intercept the network message associated with the subscription to be able to read the corresponding data, which means that the concretisation cannot be achieved using available middleware primitives alone. The other alternative, although highly inefficient is for the mediator to read all data available and publish them periodically. Prior work, both theoretical and practical, focusing on the equivalence between publish/-subscribe and tuple space middleware solutions has shown that only by adding extra functions (or 'hacks') can the two interaction patterns be coordinated [BZ01, CMP08].

To sum up, coordinating different interaction patterns is only possible in certain cases. The main issue is that, due to the increased decoupling (in space and time) between components, the mediators simply cannot synchronise with the components. Fortunately, many modern middleware solutions support several of these interaction patterns at the same time. Further investigation is also necessary in order to assert the possibility for coordination by making explicit some common 'hacks'.

5.3.2 Application to the Positioning Example

To illustrate the case of components implemented using middleware solutions based on different interaction patterns, we use a simple example from the GMES case study, the positioning example. In *Country 1*, *C2* obtains the location of firemen directly by interacting with their devices using SOAP. The firemen coming in support from *Country 2* are equipped with devices that publish their location periodically using AMQP, which is a publish/subscribe middleware solution. *C2* cannot access the position of *Country 2* firemen. A mediator is necessary to enable interoperability between *C2* and the *Country 2* firemen (noted *Positioning-B*), as depicted in Figure 5.9. This

positioning example serves as a proof of concept rather than a full case study.

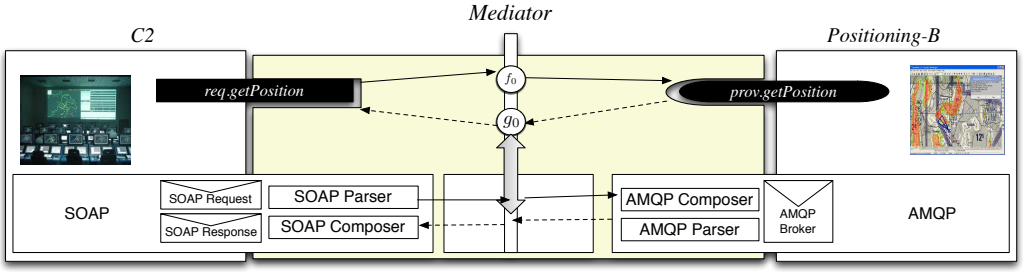

Figure 5.9. Illustrating concretisation in the positioning example

The behaviours of *C2* and *Positioning-B* are simple, as is the mediator at the application layer, but it has to be further concretised as follows:

$$
\begin{array}{l}
\mathsf{set}\ C2\_positioning\_actions = \{getPosition\} \\
\mathsf{set}\ Positioning-B\_actions = \{getPosition\} \\
\\
C2\_positioning\_role = (req.getPosition \rightarrow C2\_positioning\_role). \\
\\
Positioning\text{-}B\_role \ = (prov.getPosition \rightarrow Positioning\text{-}B\_role). \\
\\
Mediator \qquad\qquad = (c2.receiveRequest.getPosition \rightarrow pos.subscribe.getPosition \\
\qquad\qquad\quad \rightarrow pos.getEvent.getPosition \rightarrow c2.sendResponse.getPosition \\
\qquad\qquad\quad \rightarrow Mediator). \\
\\
\|PositioningSystem \ = (\ c2 : C2\_positioning\_role \\
\qquad\qquad\quad \|\ \ pos : Positioning\text{-}B\_role \\
\qquad\qquad\quad \|\ \ (\mathsf{forall}[op : C2\_positioning\_actions]\ c2 : SOAPImplementation(op)) \\
\qquad\qquad\quad \|\ \ (\mathsf{forall}[op : Positioning - B\_actions]\ pos : AMQPImplementation(op))).
\end{array}
$$

We do not specify the formal description of $\|AMQPImplementation$ as it is very similar to $\|PubSubInteraction$ since AMQP is based on the publish/subscribe interaction pattern. Although $\|PositioningSystem$ is not formally free from deadlocks, the *Positioning-B* was continually publishing the position, and the synthesised mediator was able to make *C2* and *Positioning-B* interoperate. This would be considered as a fairness requirement on the publisher.

5.4 Summary

In this chapter, we presented the steps necessary to implement the synthesised mediator. During the first step, we compute the translation functions necessary to reconcile the differences in the syntax of the input/output data used by each component. This step is sufficient to implement the mediator that ensures interoperability between components featuring differences only at the application layer. In the second step, we classified middleware solutions according to the interaction pattern they use to coordinate components' behaviours and defined an ontology per interaction pattern to make explicit the main concepts of the pattern. The two steps are sufficient to implement mediators that enable interoperability between components featuring differences at the application layer and which are based on different middleware solutions using the same interaction pattern. The third and last step consists in coordinating different interaction patterns. We showed that some transformations can be defined so as to enable the implementation of mediators that achieve interoperability between components with differences at the application layers and which are based on middleware using different interaction patterns. However, such transformations cannot always be found. Therefore, we aim to broaden this work to fully ascertain the correctness of the transformations defined and incorporate new ones.

Overall, unlike reasoning at the application layer, the correctness of the translations at the concrete level depends on the implementations involved and cannot always be proven correct. Our aim at this stage is to keep the implementation as good as a developer or an expert would perform it. We also showed that achieving interoperability from application down to middleware must be more than a localised process of accumulating mediators at the application and middleware layers.

Chapter 6

Implementation & Assessment

"Les chiffres s'impriment car ils sont clairs, on s'en souvient donc on les croit."

— in L'Art Français de la Guerre by Alexi Jenni, novelist (1963-)

A solution to interoperability can only be useful if it can easily be applied to a large number of cases. While previous chapters presented the theoretical aspects of the automated synthesis and implementation of mediators, this chapter presents its practical aspect. We describe the MICS tool that implements our approach for the automated synthesis and implementation of mediators and experiment it with different case studies, each of which highlights a feature of the tool. Through these case studies, we demonstrate the viability and wide applicability of our approach.

6.1 The MICS tool

We developed the MICS (Mediator Synthesis to Connect Components) tool to carry out the automated synthesis and implementation of mediators. This tool is available at http://www-roc.inria.fr/arles/software/mics/. MICS takes as input the models of two functionally-compatible components together with a domain ontology that represents the shared knowledge of the application domain. The model of a component is that presented in Section 3.2, which describes the interface of the component, both syntactically and semantically, and its behaviour. As output, MICS produces the concrete mediator that enables the two functionally-compatible components to interoperate.

As depicted in Figure 6.1, MICS is made up of four modules, each of which executes one step of the mediation approach:

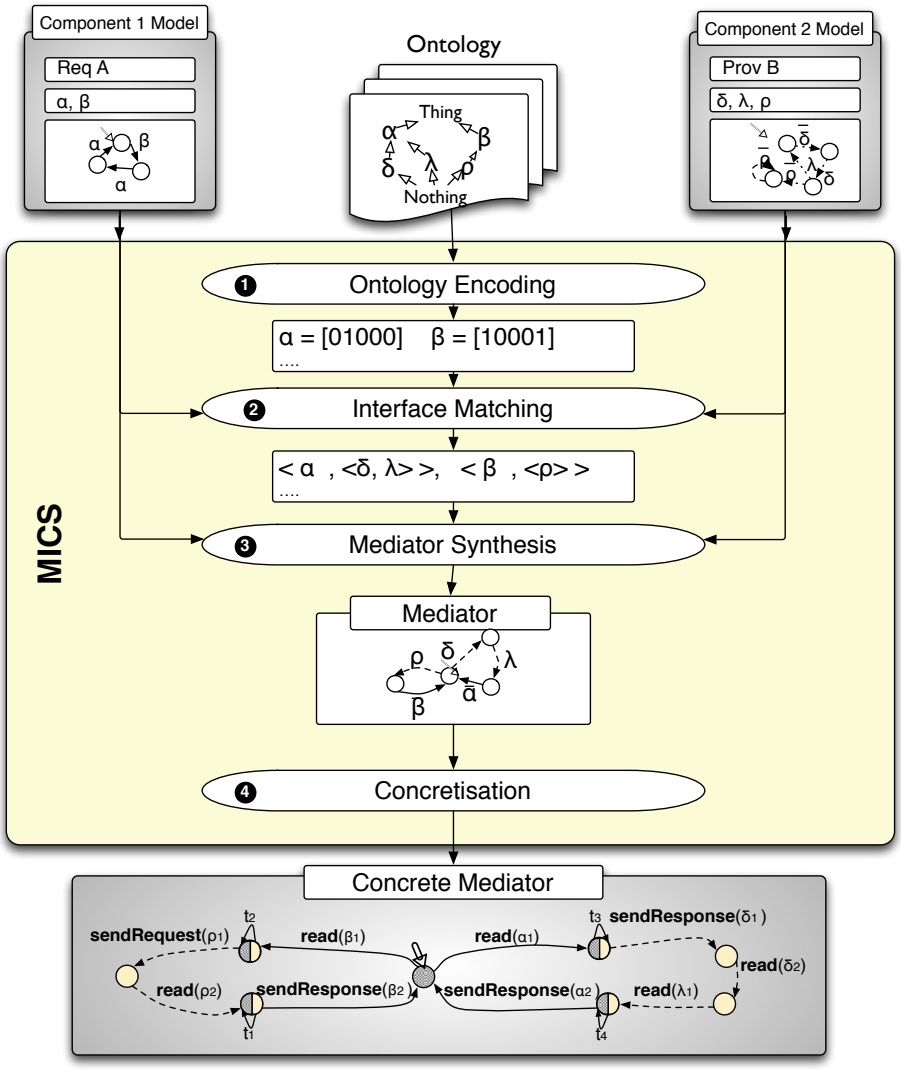

Figure 6.1. Overview of MICS

❶ **The ontology encoding module** first builds the hierarchy of the concepts of the domain ontology given as input using Pellet[1], which is an open-source Java library for OWL DL reasoning. Then, it associates a bit vector to each concept in the ontology according to Algorithm 1 in Chapter 4. Representing ontology

[1] http://clarkparsia.com/pellet/

concepts using bit vectors allows us to check subsumption between concepts using bitwise AND, while disjunction and aggregation are checked using bitwise OR. The benefit of this encoding of ontology concepts is twofold. First, we invoke the ontology reasoner just once and then simply use logical operators to verify ontological relations. Second, we can express ontological constraints using logical operators, which are supported by any constraint solver.

❷ **The interface matching module** computes the correspondences between the actions of components' interfaces. As described in Section 4.3 of Chapter 4, we formulate interface matching as a constraint satisfaction problem whose solution is obtained using Choco[1], which is an open-source Java library for constraint solving and constraint programming. We begin by encoding the interfaces using the bit vector representation of the ontological concepts. We represent the sequences of actions of each interface using a vector of variables, each of which can take a value between 0 and the cardinality of the interface. This value indicates the position of the action in the sequence, 0 meaning that the sequence does not contain the action. We allow two variables within the same sequence to have the same value to indicate that they may be executed in any order. We encode the constraints for matching sequences of actions by referring to the encoding of the interfaces. Finally, we search for all possible values that verify these constraints. Choco performs a backtracking search by selecting a value for each variable and propagating such a value throughout the search space by updating the lower and upper bounds of other variables. Constraint propagation aims to contract domains by taking all constraints into account sequentially, the goal being to reduce the search space.

❸ **The mediator synthesis module** uses the generated matchings to build a correct-by-construction mediator according to Algorithm 2 in Chapter 4. The mediator is incrementally constructed by forcing the two processes representing the behaviours of the components involved to progress synchronously so that if one requires the sequence of actions X_1, the other is ready to engage in a sequence of provided actions X_2 with which X_1 matches. The mediator also consumes extra provided actions when necessary to allow the components to progress.

[1] http://choco.emn.fr/

❹ **The concretisation module** refines the synthesised mediator by computing the translation functions necessary for reconciling the differences between the data received from one component and the data that must be sent to the other component and coordinating the components' behaviours at the middleware layer. To compute the translation functions we use, besides the ontology relations, the Harmony tool[1] for matching the XML schema representing the syntax of the data embedded in the actions of the components' interfaces. The middleware ontologies described in Chapter 5 are an integral part of the tool and are used to coordinate the components' behaviour at the middleware layer.

MICS can be used as a standalone tool or integrated in the CONNECT architecture where it plays the role of the synthesis enabler. In the former case, users select the models of two functionally-compatible components, which they can visualise. The same ontology must be used to annotate the actions of the components' interfaces and must be accessible through the URI specified in the annotations. Once the mediator is synthesised (if it exists), users can visualise the interface matchings together with the behaviour of the mediator. Finally, users can configure the deployment of the mediator. Figure 6.2 illustrates the use of MICS as a standalone tool. When integrated in the CONNECT architecture, MICS is directly invoked by the discovery enabler with the models of functionally-compatible components, and the deployment is based on the Starlink framework, as explained in Chapter 3.

6.2 Case Studies

This section reports the results of the experiments we conducted using MICS to generate mediators in different case studies, which were briefly outlined in Chapter 1. Table 6.1 summarises the differences at the application and middleware layers between the components of each case study. We begin with the instant messaging case study to illustrate one-to-one matchings. Although very similar from a behavioural aspect, instant messaging applications define different message formats that prevent users of one application from interacting with users of other applications. In addition, this case highlights the need to take into account interaction with third party components, i.e., the servers with which the users of each instant messaging application

[1] http://openii.sourceforge.net/index.php?act=tools&page=harmony

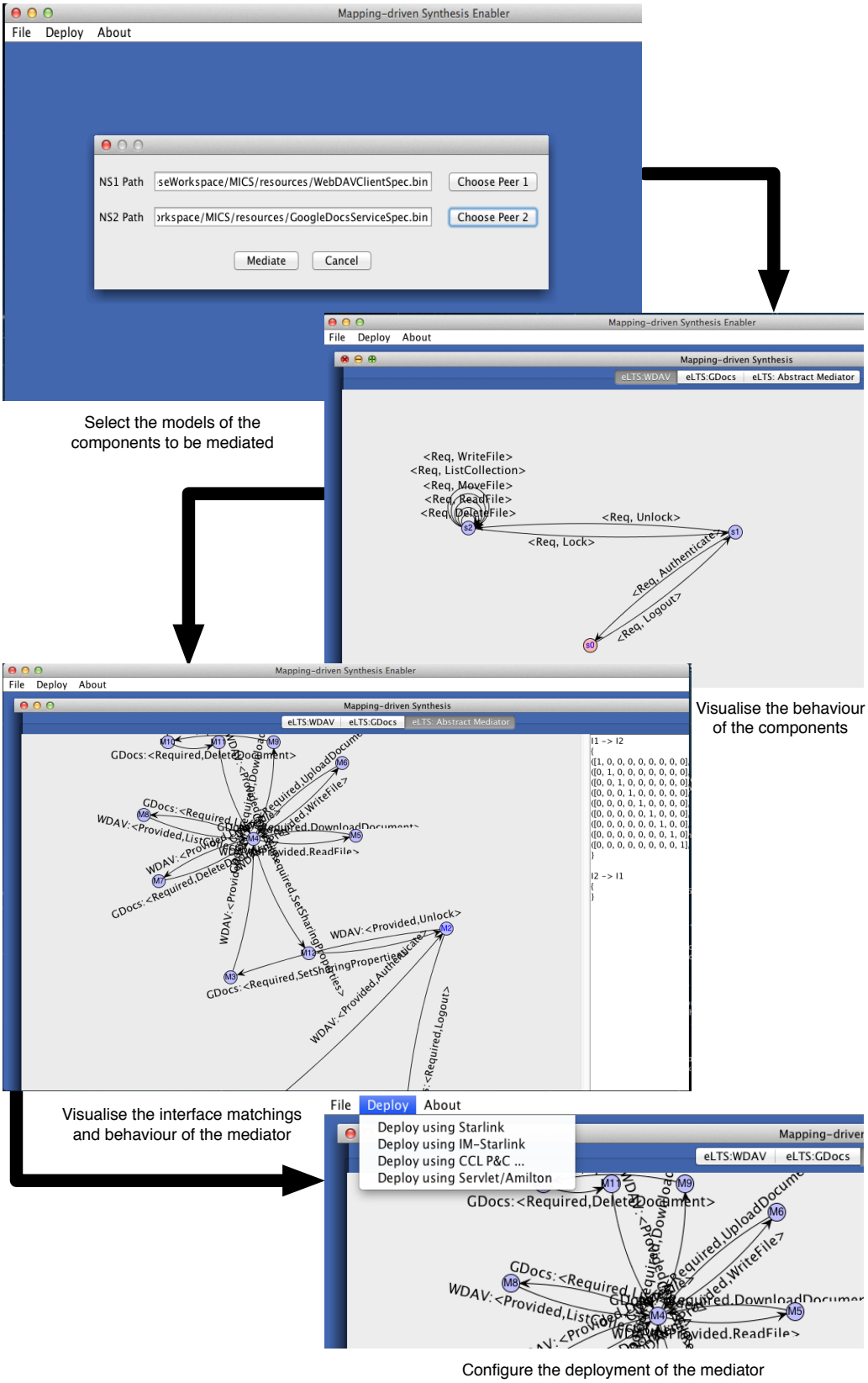

Figure 6.2. Using MICS as a standalone tool

Case Studies		Differences at the Application Layer	Differences at the Middleware Layer
Instant Messaging		one-to-one	—
File Management		one-to-many	—
Purchase Order		one-to-many & loops	—
Event Management		one-to-many	SOAP vs. HTTP (REST)
GMES	Weather	one-to-many	SOAP vs. CORBA
	Positionning	one-to-one	SOAP vs. AMQP (RPC vs. Pub/Sub)
	Vehicle	one-to-many & extra actions	—

Table 6.1. Summary of the case studies

have to login. We then move to the file management case study, which we previously used to illustrate the computation of one-to-many matchings and concretisation between components implemented using the same middleware. The third case study is the Purchase Order Mediation scenario defined in the context of the Semantic Web Service (SWS) Challenge [PMLZ08]. This case study has been the subject of extensive investigation in the SWS domain and hence it permits us to evaluate our solution for the synthesis of mediators against several mediation solutions from the SWS domain. The fourth case relates to interoperability between event management systems, which are concerned with the organisation of events like conferences, seminars, concerts. Besides one-to-many matchings at the application layer, this case illustrates differences at the middleware layer as the components involved were built using SOAP and REST Web Services respectively. Finally, we present the experiments we conducted within the European CONNECT project using GMES. The aim of the GMES experiment is to illustrate our solution in the context of systems of systems and highlight its integration in the CONNECT approach for eternal interoperability. GMES requires achieving interoperability between components that feature differences at the application and middleware layers, including dealing with different interaction patterns, namely RPC and publish/subscribe. GMES also demonstrates the need for mediation at runtime between components that are dynamically discovered and whose models are automatically completed using learning techniques. In the following, we describe each case study in more detail. Whenever applicable, we

present the ontology, the models of the components involved and the performance of the synthesised mediator.

6.2.1 Instant Messaging: One-to-One Matching

The first case study relates to interoperability between different Instant Messaging (IM) applications. Popular and widespread IM applications include Windows Live Messenger[1] (commonly called MSN Messenger), Yahoo! Messenger[2], and Google Talk[3], which is based on the XMPP[4] standard protocol. These IM applications offer similar functionalities such as managing a list of contacts or exchanging textual messages. However, a user of MSN Messenger is unable to exchange instant messages with an XMPP user (see Figure 6.3). As a result, users have to either install many applications or to use third party applications (e.g., Pidgin[5] or Adium[6]) to be able to chat with one another. This situation, though cumbersome from a user's perspective, unfortunately reflects the way IM —like many other existing applications— has developed. Our aim is to let users install their favourite IM applications and synthesise a mediator that performs the necessary translations to make different IM applications interoperable. Each IM application represents a component in the composed IM system, which can only achieve users' requirements to exchange messages if the two components can interoperate.

Figure 6.3. Making XMPP and MSNP components interoperable

Even though IM components are simple and quite similar from a behavioural point of view, each one defines its own message format. For example, Windows Live Messenger (MSNP) uses text-based messages whose structure includes several constants with predefined values whereas Yahoo! Messenger (YMSG) uses binary

[1] http://explore.live.com/windows-live-messenger/
[2] http://messenger.yahoo.com/
[3] http://www.google.com/talk/
[4] Extensible Messaging and Presence Protocol – http://www.xmpp.org/
[5] http://www.pidgin.im/
[6] http://adium.im/

messages that include a header and key-value pairs. As for XMPP messages, they are defined according to a given XML Schema.

The Instant Messaging Ontology

To enable the automated synthesis of mediators between IM components, we started by defining the IM ontology depicted in Figure 6.4. An InstantMessage has at least one sender hasSender, one recipient hasRecipient, and one message hasMessage. hasSender and hasRecipient are object properties that relate an instant message to a sender or a recipient while hasMessage is a data property that specifies the content of an instant message. Sender and Recipient are both subsumed by the User. A Conversation is performed between a sender (who initialises it) and a recipient, and the conversation has its own identifier. A Conversation is an aggregation of instant messages. A ChatRoom represents a venue that multiple users can join to exchange messages. The IM ontology also represents the operations used by the IM components such as Authentication and Logout. Note though that while both MSN_Authentication and XMPP_Authentication are subsumed by Authentication, there is not any direct relation between them. The same is true for Logout.

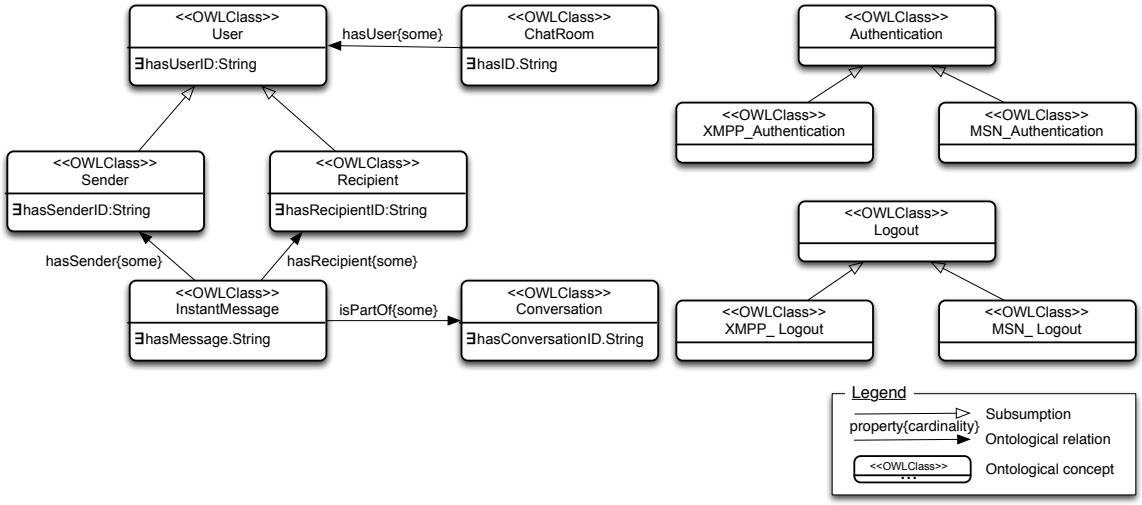

Figure 6.4. The IM ontology

Behavioural Specifications of the Instant Messaging Components

We used the IM ontology to annotate the interfaces of the two IM components and describe their behaviours as follows:

$$
\begin{aligned}
\mathit{MSNP} &= (<\mathsf{MSN\_Authentication\_Request}, \{\mathsf{UserID}\}, \{\mathsf{Challenge}\} > \\
&\quad \rightarrow <\mathsf{MSN\_Authentication\_Response}, \{\mathsf{Response}\}, \{\mathsf{Authentication\_ok}\} > \\
&\quad \rightarrow \mathsf{ExchangeMsgs}), \\
\mathsf{ExchangeMsgs} &= (<\mathsf{CreateChatRoom}, \{\mathsf{UserID}\}, \{\mathsf{ConversationID}\} > \\
&\quad \rightarrow <\mathsf{JoinChatRoom}, \{\mathsf{UserID}\}, \{\mathsf{Acceptance}\} > \rightarrow \mathsf{P1} \\
&\quad | < \overline{\mathsf{JoinChatRoom}}, \{\mathsf{UserID}\}, \{\mathsf{Acceptance}\} > \\
&\quad \rightarrow < \{\mathsf{ChatRoomInfo}, \emptyset, \{\mathsf{ConversationID}\} > \rightarrow \mathsf{P1}), \\
\mathsf{P1} &= (<\mathsf{InstantMessage}, \{\mathsf{UserID}, \mathsf{ConversationID}, \mathsf{Message}\}, \emptyset > \rightarrow \mathsf{P1} \\
&\quad | <\overline{\mathsf{InstantMessage}}, \{\mathsf{UserID}, \mathsf{ConversationID}, \mathsf{Message}\}, \emptyset > \rightarrow \mathsf{P1} \\
&\quad | <\mathsf{MSN\_Logout}, \{\mathsf{UserID}\}, \emptyset > \rightarrow \mathsf{END}).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{XMPP} &= (< \mathsf{XMPP\_Authentication\_Request}, \{\mathsf{UserID}\}, \{\mathsf{Challenge}\ \} > \\
&\quad \rightarrow <\mathsf{XMPP\_Authentication\_Response}, \{\mathsf{Response}\}, \{\mathsf{Authentication\_ok}\} > \\
&\quad \rightarrow \mathsf{ExchangeMsgs}), \\
\mathsf{ExchangeMsgs} &= (<\mathsf{InstantMessage}, \{\mathsf{SenderID}, \mathsf{RecepientID}, \mathsf{Message}\}, \emptyset > \rightarrow \mathsf{ExchangeMsgs} \\
&\quad | <\overline{\mathsf{InstantMessage}}, \{\mathsf{SenderID}, \mathsf{RecipientID}, \mathsf{Message}\}, \emptyset > \\
&\quad \rightarrow \mathsf{ExchangeMsgs} \\
&\quad | < \mathsf{XMPP\_Logout}, \{\mathsf{UserID}\}, \emptyset > \rightarrow \mathsf{END}).
\end{aligned}
$$

Each IM component performs authentication and logout with the associated server. Before exchanging messages, the MSNP component has to configure a *chat room* where the MSN conversation can take place between the user that initiates this conversation (sender) and the user who accepts to participate in this conversation (recipient). XMPP simply includes the sender and the recipient identifiers in each message.

The Instant Messaging Mediator

There are only one-to-one matchings between the actions of IM components. If we consider, for example, the action α=<InstantMessage, {UserID, ConversationID, Message}, $\emptyset$> required by MSNP and $\overline{\beta}$=<$\overline{\text{InstantMessage}}$, {SenderID, RecipientID, Message}, $\emptyset$> provided by XMPP. The IM ontology indicates that: (i) Sender is subsumed by User, and (ii) ConversationID identifies a unique Conversation, which includes InstantMessage that is associated with

Recipient identified by a RecipientID attribute. Consequently, $\alpha \mapsto \overline{\beta}$. Note however that some actions, e.g., MSN_Authentication_Request, XMPP_Authentication_Request, or JoinChatRoom, are not subject to mediation as they are exchanged with proprietary servers rather than between the two components. The mediator simply translates the InstantMessage exchanged between the components. For the implementation of the mediator, we used Starlink as it allows us to manage proprietary protocols. However, we were faced with two problems. First, IM components do not support dynamic discovery or binding but directly exchange messages with the corresponding server. Nevertheless, all of them support the configuration of a proxy that can be used as an intermediary. As a result, we had to configure the proxy settings of each IM component with the address of the computer where Starlink, which executes the mediator, is deployed. Second, the list of contacts is sent directly to the IM component by the corresponding server and cannot include users from other IM systems. More specifically, while the mediator can enable the IM user to add a contact, it cannot manage the persistence of contact lists by itself. So we manually added the software code necessary to intercept and save lists of contacts across systems.

Once the mediator has been synthesised and the IM components configured, we measured the time required to exchange a 100-character message between any com-

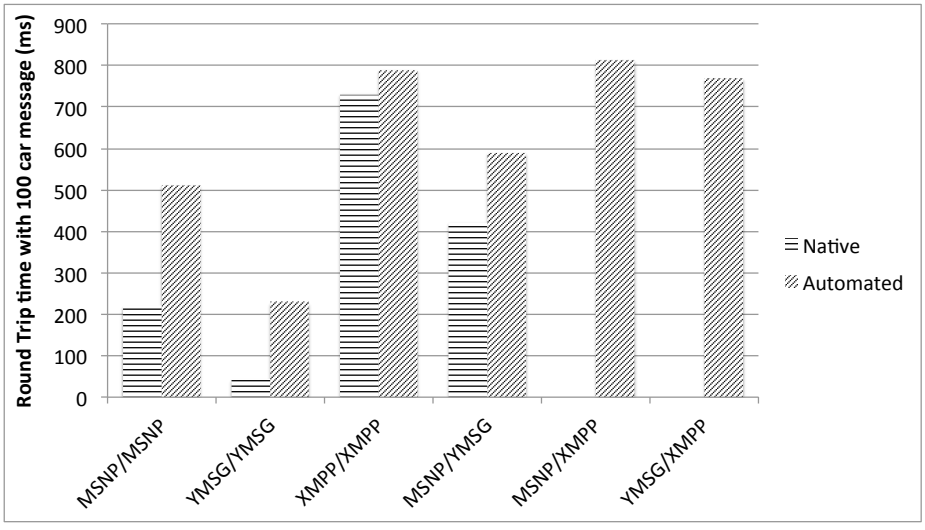

Figure 6.5. Latency for mediated and non-mediated interactions between IM components

bination of MSNP, YMSG, and XMPP. We repeated the experiments 50 times and report the average time in Figure 6.5. Besides native interactions between the IM components, we also have native interactions between MSNP and YMSG through a proprietary gateway, which was developed following an agreement between Yahoo and Microsoft [Mic10]. To distinguish between the overhead due to the mediator itself and that related to the parsers and composers, we deployed a mediator between the IM components that are already interoperable. The mediator in this case simply receives and sends back the message unchanged. The overhead introduced by parsers and composers depends on the encoding of the instant messages: while the overhead is negligible for the XML-based XMPP components, it is significant in the case of the binary YMSG protocol. Nevertheless, guidelines for response time in interactive applications specify that 1s is the limit to keep the user's flow of thought seamless [Nie93]. Hence, the overhead of the mediator execution is acceptable. Furthermore, the mediator itself only introduces negligible overhead, hence the latency in the case of MSNP and YMSG interacting with XMPP remains close to that of XMPP.

6.2.2 File Management: One-to-Many Matching

This case study relates to interoperability between WebDAV and Google Docs. It was used in Chapters 4 and 5 to illustrate the synthesis of mediators and their implementation in the case of differences between components at the application layer only, as both components are based on HTTP. In this section, we evaluate the performance of the generated mediator.

We deployed the mediator over an Apache Tomcat[1] container in order to intercept and filter out the HTTP messages. We measured the time it took to perform a simple conversation consisting of an authentication, moving a file from one folder to another, and listing the content of the two folders. As for performance measurements, the file is a 4KB text document to lessen the network delay. We used a WebDAV client developed using the Sardine library[2] and our enterprise WebDAV repository. Note that we accessed our enterprise WebDAV repository via the Internet since the Google Docs service can only be invoked via the Internet. We repeated each conversation 50

[1] http://tomcat.apache.org/
[2] http://sardine.googlecode.com/

times and report the average execution time in Figure 6.6. In the case of the WebDAV client interacting with the Google Docs Service, the overhead is negligible compared to the Google Docs interactions, while it is 75% more than the WebDAV service. This is mainly due to the communication time since the former requires 3 actions (i.e., 6 messages) while the latter requires only one action (i.e., 2 messages). In the case of a Google Docs client interacting with the WebDAV service, the mediator introduces an 18% time overhead compared to native Google Docs interactions while it is twice the time compared to native WebDAV interactions. This increase in time is due to the fact that the Google Docs actions are translated on a one-to-one basis, that is a DownloadDocument to a ReadFile, UploadDocument to WriteFile, and DeleteDocument to DeleteFile, and not merged into one MoveFile action. In other words, when the Google Docs client performs the three actions to move a file, then these actions are translated one by one and not merged into a single action. Such an optimisation is however hard to predict as the behaviour of the Google Docs client specifies that these three actions can be performed in any order.

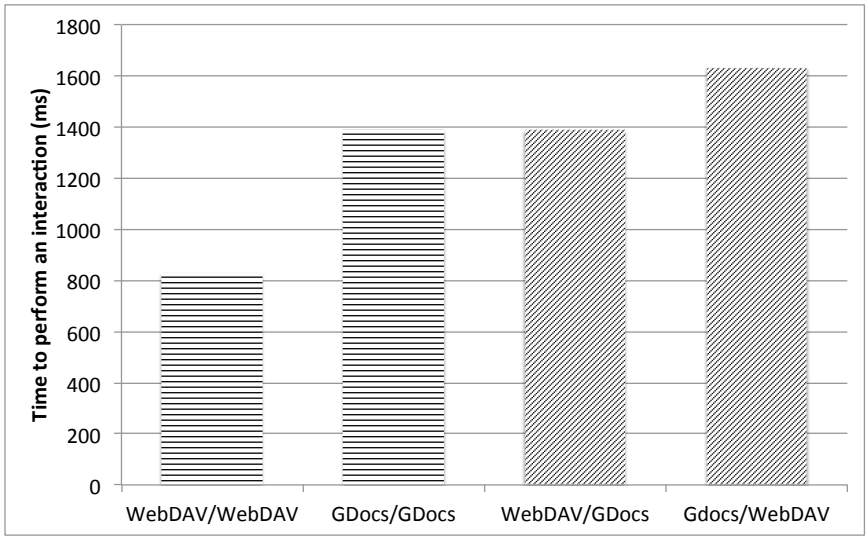

Figure 6.6. Latency for mediated and non-mediated interactions between WebDAV and Google Docs

6.2.3 Purchase Order: Mediation of Semantic Web Services

The Purchase Order Mediation scenario [PMLZ08] has been proposed in the context of the SWS challenge to provide a common ground to discuss semantic (and other) Web Service solutions and compare them according to the set of features that a mediation solution should support.

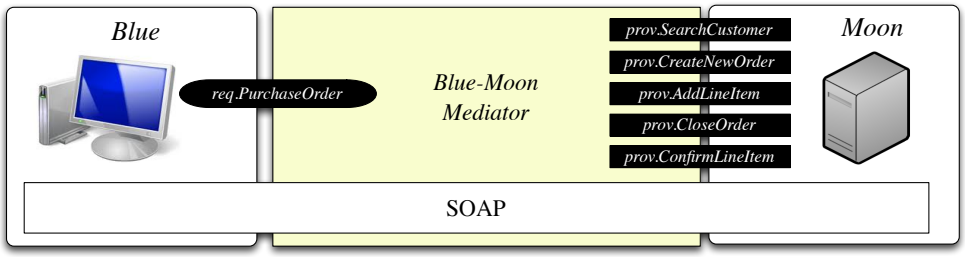

Figure 6.7. Making *Blue* and *Moon* interoperable

The scenario highlights the need for mediation in the eBusiness domain. It describes a Web Service client, called *Blue*, that is used to purchase some product from a Web Service, called *Moon*. However, although *Moon* provides the functionality required by *Blue*, their interfaces and behaviours are different as they were developed independently (see Figure 6.7). In particular, *Blue* uses a single action to make its purchase. The input of this action contains all the information necessary for placing the order including the customer information and the list of items that the customer wishes to purchase. On the other hand, *Moon* provides many actions to perform the same task; customer must get an identifier, create an order, add items to the order, and perform a confirmation for each items. Hence, a mediator is needed to enable *Blue* and *Moon* to interoperate.

The Purchase Ontology

We build upon the purchase order ontology, which is publicly available as part of the WSDL-S specification[1], and we extend it in order to include concepts representing the operations of *Blue*'s and *Moon*'s interfaces. Figure 6.8 shows an extract of the purchase ontology. The ontology shows the relations holding between the various concepts used by two purchase order components. It specifies the attributes of

[1] http://www.w3.org/Submission/WSDL-S/

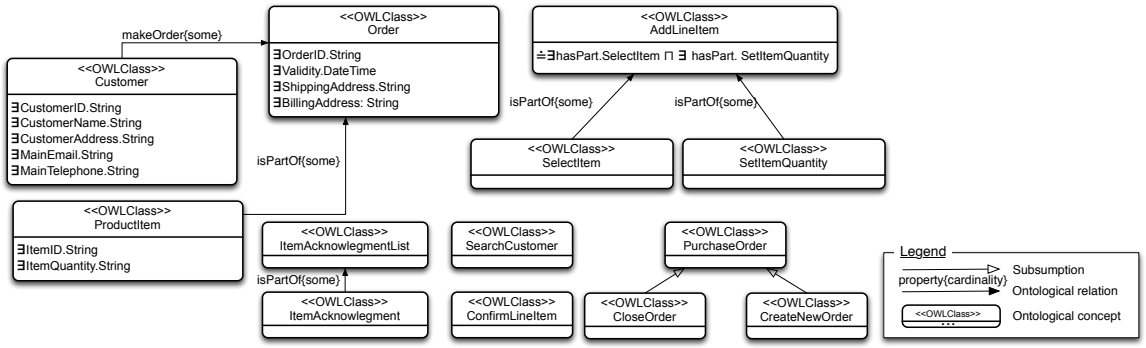

Figure 6.8. The purchase ontology

each concept; for example Order is characterised by several data properties: OrderID, ShippingAddress, and Billing defined as strings and Validity specified using the OWL built-in dateTime type. An order contains one or many ProductItem, each of which is associated with an identifier and some quantity. The operation AddLineItem is defined as the aggregation of the SelectItem and SetItemQuantity.

Behavioural Specifications of the Purchase Order Components

Blue initiates the purchasing process by sending the customer information and the list of products he wants to purchase. *Blue* expects an acknowledgement which confirms the items that can be delivered. Its behavioural description is as follows:

$$Blue = (<\mathsf{PurchaseOrder},\ \{\mathsf{CustomerName}, \mathsf{ShippingAddress}, \mathsf{BillingAddress}, \mathsf{ItemList}\}, \{\mathsf{ItemAcknowledgmentList}\} >\!\!\rightarrow \mathsf{END}).$$

Moon decouples the management of customers from the management of orders. To use *Moon*'s service, clients must first retrieve relevant customer details including his identifier. *Moon* also verifies that the customer is authorised to create a new order. The customer can then add items to the newly created order. Once all the items have been added, the customer closes the order and asks *Moon* to confirm the delivery of each item. Its behavioural specification is the following:

$$
\begin{aligned}
\mathit{Moon} = (&<\overline{\mathsf{SearchCustomer}},\ \{\mathsf{CustomerName}\},\ \{\mathsf{Customer}\} > \\
&\rightarrow <\overline{\mathsf{CreateNewOrder}},\ \{\mathsf{CustomerID}, \mathsf{ShippingAddress}, \mathsf{BillingAddress}\}, \\
&\quad \{\mathsf{OrderID}\} > \rightarrow <\overline{\mathsf{AddLineItem}},\ \{\mathsf{OrderID}, \mathsf{ItemID}, \mathsf{ItemQuantity}\},\ \emptyset > \rightarrow \mathsf{P1}), \\
\mathsf{P1} \quad = (&<\overline{\mathsf{AddLineItem}},\ \{\mathsf{OrderID}, \mathsf{ItemID}, \mathsf{ItemQuantity}\},\ \emptyset > \rightarrow \mathsf{P1} \\
&\mid <\overline{\mathsf{CloseOrder}},\ \{\mathsf{OrderID}\},\ \{\mathsf{OrderAcknowledgment}\} > \rightarrow \mathsf{P2}), \\
\mathsf{P2} \quad = (&<\overline{\mathsf{ConfirmLineItem}},\ \{\mathsf{OrderID}, \mathsf{ItemID}\}, \{\mathsf{ItemAcknowledgment}\} > \rightarrow \mathsf{P2} \\
&\mid <\overline{\mathsf{ConfirmLineItem}},\ \{\mathsf{OrderID}, \mathsf{ItemID}\}, \{\mathsf{ItemAcknowledgment}\} > \rightarrow \mathsf{END}).
\end{aligned}
$$

The SWS Challenge provides relevant information about the components using WSDL and natural language descriptions. *Blue* and *Moon* are provided by the SWS Challenge organisers and cannot be altered, although their descriptions may be semantically enriched. We attached semantic annotations to the interfaces of *Blue* and *Moon*. We specified the behaviour of each component based on the textual descriptions and sequence diagrams given on the SWS Challenge Web site[1] and the related book [PMLZ08]. Even though there are techniques to associate semantic annotations to the interface description of a component or to extract the associated behaviour, we did it manually since we focus on mediation between components rather than on the inference of the semantic annotations or the behaviour of a component. Furthermore, during the SWS Challenge participants were asked to extend the syntactic descriptions in such a way that their solutions could perform the necessary mediation.

Comparison with Solutions for the Purchase Order Mediation Scenario

In the following, we compare our solution for automated synthesis of mediators to those presented at the SWS challenge. Table 6.2 summarises this comparison.

First, Brambilla *et al.* [BCV+08] define the SWE-ET framework, which eases the description of the components as well as the refinement and deployment of the mediator. SWE-ET facilitates the implementation and maintenance of the mediator by following a software engineering process rather than generating the mediator automatically. Our aim is rather to generate the mediator automatically while relying on existing tools to annotate the interfaces of the components and define the associated behaviour, and also to execute the mediator.

Vitar *et al.* [VZMM08] build upon the WSMO conceptual model for mediation,

[1] http://SWSChallenge.org/wiki/

which includes the use of WSML to describe components and WSMX for the execution of mediators. The authors assume one ontology per component instead of using a common ontology to annotate both components. We agree that the components may be defined using heterogeneous ontologies, but in this case subsumption between concepts is associated with some confidence score, and the synthesis algorithm should deal with the varying confidence about concept relations and manage the imprecision involved. Rather, the WSMO solution assumes the translation to be manually specified at design time. Unlike WSMO, we use the ontology to reason about the semantics of the actions of components' interfaces and compute the translations automatically. Furthermore, although the schema translations cannot be proved correct in theory, it turns out that the computed translation function was sufficient to translate the data exchanged between *Blue* and *Moon*.

Kubczak *et al.* [KMSN08] use process algebra to model the behaviour of *Blue* and *Moon*, utilise constraints to ensure safety and consistency of the mediation, and abstraction/concretisation to deploy the mediator model using various communication standards. The authors assume that the user specifies an SLTL (Semantic Linear-time Temporal Logic) formula that describes the goal that needs to be fulfilled by the system, based on which they compute the mediator. The use of goals can indeed facilitate the synthesis of mediators but it also raises other issues about the management of these goals: users have to write an SLTL formula to make use of the components. We believe that this might be a somewhat restrictive requirement.

Finally, Gomadam *et al.* [GRW+08] augment component descriptions with precondition/effects and use planning techniques to compute the mediator. First the authors specify that it is certainly not enough to consider only precondition/effect (as is usually done with planning algorithms) but also the input/output data as well. However, they do not specify how this should be performed. Another drawback of the approach, although not visible from the challenge example, is that it only works with a unique one-to-many matching and cannot handle interactions where both components require and provide actions, as is the case in peer-to-peer interactions.

While the Purchase Order Mediation Scenario provides us with a common case study to evaluate our solution for the synthesis of mediators against several solutions from the SWS domain, we cannot evaluate the implementation of mediators. Indeed, we cannot evaluate the performance of the generated mediators, as the services involved no longer maintained.

Solution	Key Features	Similarities	Differences
SWE-ET [BCV+08]	• An MDE approach for the design and code generation of the mediator • Semi-automated specification and refinement of the mediator using WebML/BPMN	• Combined data and behavioural mediation	• Mediator specified by the developer • Data mediation through lowering and lifting
WSMX [VZMM08]	• Build upon the WSMO ontology to specify components and reason about their interoperability	• Combined data and behavioural mediation	• Use of many ontologies • Data mediation through lowering and lifting • No guarantee about deadlock freeness
Kubczak *et al.* [KMSN08]	• Mediator extracted from the goal • Model checking to verify safety properties on the mediator	• Use of process algebra • Use of constraints to ensure safety	• SLTL formula to define the goal • Constraints defined manually
Gomadam *et al.* [GRW+08]	• Planning algorithms to synthesise the mediator	• Combined data and behavioural mediation	• Use of preconditions and effects in component specifications • Data mediation through lowering and lifting

Table 6.2. Comparing our automated mediation solution with the solutions participating in the SWS challenge

6.2.4 Event Management: Unified Application-Middleware Mediation

In this case study we consider mediation in the event management domain. Event management systems provide various services such as ticketing, attendee management, and payment for various events like conferences, seminars, concerts, etc. Event organisers usually rely on existing and specialised event management systems to prepare their events as they generally offer an economical and better quality solution compared to building their own system. In order to use an event management service, an organiser has to include within its application a client implementing the API defined by the service provider.

However, depending on the event they are in charge of, organisers may have to integrate with multiple event management providers. Event management systems often exhibit different APIs and are implemented using different middleware technologies. Our solution intends to simplify and automate the support of multiple services belonging to the same application domain, e.g., event management. We investigated this case study with an industrial partner Ambientic[1]. Ambientic provides a suite of services to facilitate the organisation of events (e.g., conferences, trade shows, and exhibitions) and improve collaboration between the different stockholders (organisers, visitors, and speakers). Ambientic expressed the need to interface with existing event management services as required by the customers. However, while developing one client is feasible, developing a client for each event management service rapidly becomes fastidious.

We consider that a client for Amiando[2] has been developed and synthesise a mediator that enables this client to interact with the RegOnline[3] service as depicted in Figure 6.9. The Amiando client is developed according to the REST architectural style, uses HTTP as the underlying communication protocol, and relies on JSON[4] for data formatting. On the other hand, the RegOnline service is implemented on top of SOAP, which implies using WSDL[5] to describe the application interface, and is further bound to the HTTP protocol.

[1] http://www.ambientic.com/en/
[2] http://developers.amiando.com/
[3] http://developer.regonline.com/
[4] http://www.json.org
[5] http://www.w3.org/TR/wsdl

Figure 6.9. Making Amiando client and RegOnline service interoperable

In addition, these differences at the middleware layer, Amiando and RegOnline also differ at the application layer. 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 the `String` type. The RegOnline GetEvents operation has a `Filter` argument used to specify the keywords to search for and which is also of the `String` type. However, the RegOnline developer documentation specifies that this string is in fact a C# expression and can contain some .NET framework method call (such as `Title.contains("keyword")`), which is incompatible with the Amiando search string. Regarding behavioural differences, 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. First, we perform EventFind to get a list of conference identifiers. Then, for each element of the list we call the EventRead operation with the identifier as a parameter to get information about the conference.

The Event Management Ontology

We built upon the eBiquity event ontology[1], which defines the vocabulary for describing and relating information elements that are commonly used in event management systems. Figure 6.10 shows an extract of this ontology once the classification has been performed. Event is described using several attributes (e.g., identifier, start and end dates, and title), each of which is represented as a data property. EventList is the aggregation of many events. The eBiquity event ontology does not define the semantics of operations, which we added ourselves. GetEvent is described as the aggregation of EventFind and EventRead.

[1] http://ebiquity.umbc.edu/ontology/event.owl

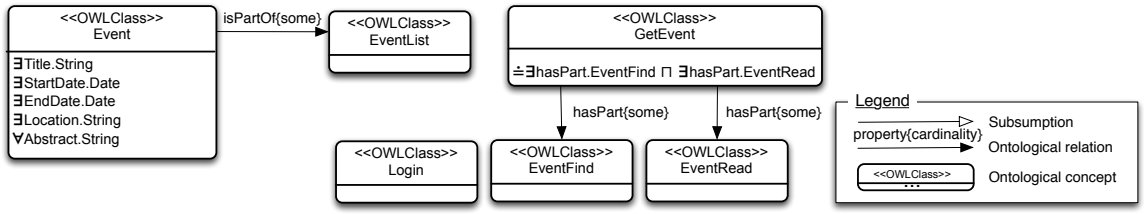

Figure 6.10. The event management ontology

Behavioural Specifications of the Event Management Components

The interfaces of Amiando and RegOnline include more than 50 operations each. Therefore and to make the presentation easier, we consider a simplified scenario. Still, we strictly follow the publicly available APIs of the two providers. In this scenario, the client searches for events that include some keywords in their title, and then examines the information about the found events. In Amiando, the clients have to send an EventFind request with the keywords to search for. Any Amiando client is given a unique and fixed `ApiKey`, which is an authentication token that must be included in any interaction with the Amiando service. The EventFind response includes a list of event identifiers. To get the information about an event, clients issue an EventRead request with the event identifier as a parameter. In RegOnline, clients must first perform a Login action and get an authentication token represented by the `ApiToken` field, which must be included in the following requests. After that, they send a GetEvents request, which includes a `Filter` argument specifying the keywords to search for. In return, the client gets the list of events, each with the associated information, verifying the search criteria.

The behavioural specifications of the components are as follows:

$$
\begin{aligned}
&\mathit{Amiando} = (<\mathsf{EventFind}, \{\mathsf{AuthentificationToken}, \mathsf{Title}\}, \{\mathsf{EventIDList}\} >\!\!\rightarrow \mathsf{P1}),\\
&\mathsf{P1} \quad = (<\mathsf{EventRead}, \{\mathsf{EventID}\}, \{\mathsf{Event}\} >\!\!\rightarrow \mathsf{P1}\\
&\qquad\quad \mid <\mathsf{EventRead}, \{\mathsf{EventID}\}, \{\mathsf{Event}\} >\!\!\rightarrow \mathsf{END}).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{RegOnline} = (<\overline{\mathsf{Login}}, \{\mathsf{Username}, \mathsf{Password}\}, \{\mathsf{AuthentificationToken}\}>\!\!\rightarrow \mathsf{P1}),\\
&\mathsf{P1} \quad = (<\overline{\mathsf{GetEvent}}, \{\mathsf{Title}\}, \{\mathsf{EventList}\}>\!\!\rightarrow \mathsf{END}).
\end{aligned}
$$

The Event Management Mediator

As well as synthesising a mediator between the Amiando client and the RegOnline service, we also synthesised a mediator between the RegOnline client and the Amiando service. In both cases we performed a simple conversation, which consists of a search based on a substring of the title of events, then getting a list of 10 events with the corresponding description for each event. We repeated each conversation 30 times and report the average execution time in Figure 6.11. Even natively, Amiando necessitates 3 times more time than RegOnline. This is due to the fact that it must send several messages on the network, each of which contains the information of one event whereas in RegOnline, the description of all events is sent in a single message. This is reflected in the case of RegOnline client interacting with the Amiando service as the mediator has to make many requests in order to create the message required by the client. One way to remedy this overhead is to send requests in parallel, but the synthesised mediator is not equipped for that and performs the translations only in sequence. For an Amiando client, using the RegOnline service is even more efficient than using its own service because the mediator invokes RegOnline once and keeps the results; hence when the Amiando client sends an EventRead the response is ready and no extra processing is necessary.

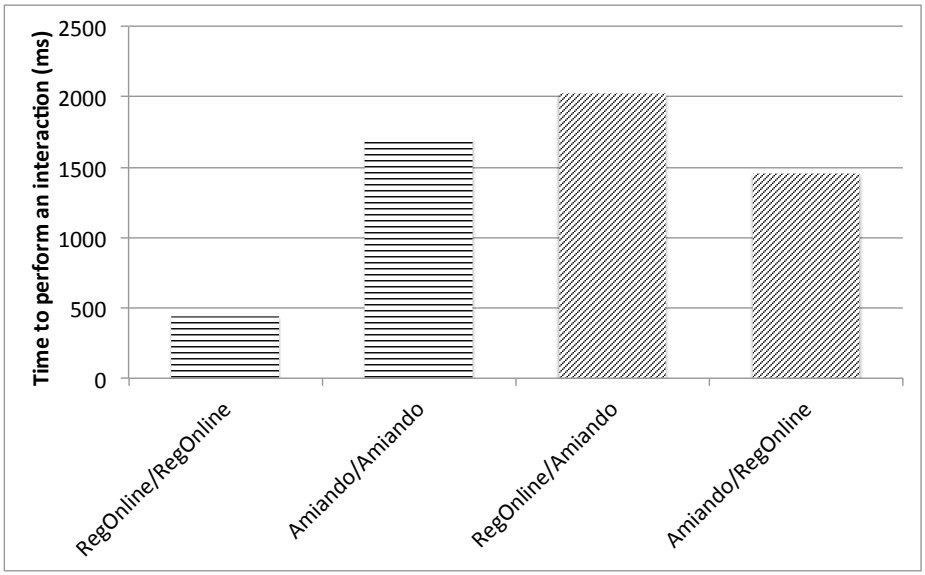

Figure 6.11. Latency for mediated and non-mediated interactions between Amiando and RegOnline

6.2.5 GMES: Runtime Mediation

We conclude the analysis with the GMES case [Con12], which we use mainly to illustrate integration within the CONNECT approach and to highlight the need for mediation at runtime in the context of systems of systems. GMES further exemplifies interoperability between components relying on middleware that implements heterogeneous interaction patterns. In this experiment, rather than specifying the component models manually, the discovery enabler is used to locate the components and complete their models by invoking the learning enabler. The synthesis of mediators is also triggered by the discovery enabler when it identifiies functionally-compatible components.

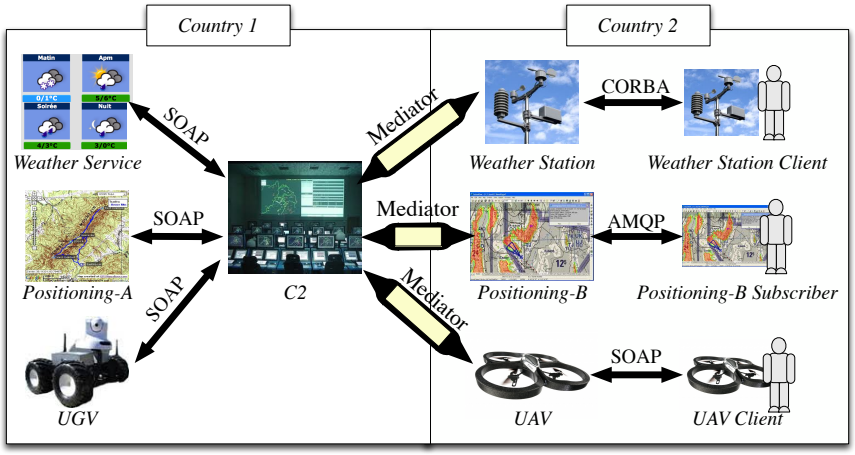

Figure 6.12. Illustrating interoperability in GMES

Figure 6.12 depicts the GMES case study. Besides mediation between *C2* and either *Weather Station* or *Positioning-B*, which we described in Chapter 5, we also need to mediate interactions between *C2* and *UAV* (Unmanned Aerial Vehicle). In this latter case, *C2* natively interacts with *UGV* (Unmanned Ground Vehicle) using SOAP. *UAV* also uses SOAP but differs from *UGV* at the application layer: *UGV* requires the client to login, then it can move in the four cardinal directions whereas *UAV* is required to take off prior to any operation and to land before logging out, which is the case of extra provided actions. We recall that the components are dynamically discovered and their interactions take place at runtime without a priori knowledge about their respective interfaces and behaviours. As a result, to enable *C2* to use the components provided by *Country 2*, with which it is functionally

compatible, mediators have to be synthesised and deployed at runtime.

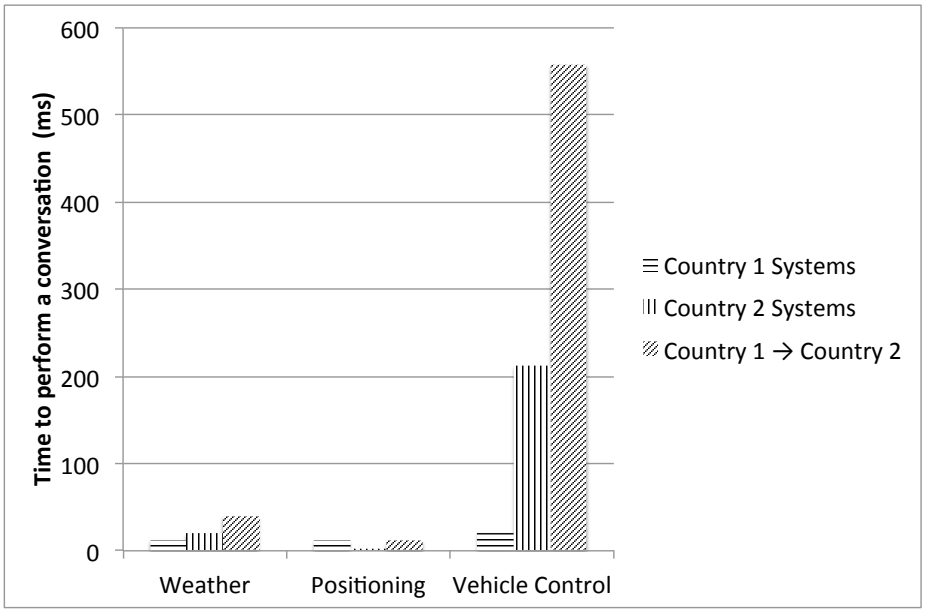

Figure 6.13. Latency for mediated and non-mediation interaction between GMES components

For each example, we measured the average time necessary to perform a meaningful conversation. In the weather example, the conversation includes authentication, obtaining weather information using a single *getWeather* operation in *Country 1* and two operations, *getTemperature* and *getHumidity* in *Country 2*, and then logging out. In the positioning example, a single operation *getPosition* is performed. In the vehicle control example, conversations consist in authentication, takeoff (in the case of *UAV* only), moving and turning both left and right, landing (in the case of *UAV* only), and logging out. We repeated each conversation 50 times and computed the average duration. The results are presented in Figure 6.13. We can see that in the case of positioning, the overhead is almost non-existent since there is no processing time on the server as the data are already published. In the weather example, the mediated conversation takes twice the time required to interact with the service using the native client. This is due to the use of Starlink underneath. While Starlink provides a very flexible and generic approach for the generation of parsers and composers, it comes with a considerable performance cost. This is one of the reasons that led us to reuse existing libraries to generate parsers and composers that, although

they are less generic, provide better performance. This problem is exacerbated in the case of vehicle control where mediated conversations require 2.6 times more time to be performed. The reason is the way that Starlink deals with extra actions (*takeoff* and *land*).

6.3 Performance of MICS

In the previous section, we showed that automatically synthesised mediators enable heterogeneous components to interoperate while introducing an acceptable overhead. In Chapter 4, we proved that the synthesised mediator guarantees that the composed system is deadlock-free. Let us now consider the time taken to synthesise mediators. Table 6.3 summarises the time to perform each mediation step —ontology encoding, interface matching, and synthesis of correct-by-construction mediators— for the aforementioned case studies.

	MSNP-XMPP	*WDAV-GDocs*	*Blue-Moon*	Amiando-RegOnline	Weather	GMES Positioning	Vehicle
Number of concepts (Disjunctions)	10 (0)	78 (7)	34 (2)	8 (2)	283 (2)	283 (2)	283 (2)
Time for encoding (ms)	300	2502	1476	834	9689	9689	9689
$\|I_1\| \times \|I_2\|$	9×5	9×7	1×5	2×2	3×4	1×1	7×11
Time for matching (ms)	37	672	481	247	342	20	567
$\|States(P_1)\| \times \|States(P_2)\|$	7×4	3×2	2×6	2×3	2×2	1×1	2×4
Time for synthesis (ms)	4	5	5	5	2	<1	7

Table 6.3. Processing time (in milliseconds) for each mediation step

First, the time for ontology encoding mainly depends on the size, i.e., the number of concepts, of the ontology. The greater the size of the ontology, the more time it takes to encode it. The number of disjunctions also influences the encoding, for example the ontology used for event management includes 8 concepts while the IM ontology includes 10 concepts but it takes 3 times longer to encode the former as it also includes disjunction concepts. The time to perform interface matchings depends mainly on the type of matching encountered. In the case of one-to-one matching, this time is minimal even with larger interfaces. In fact, computing one-to-one matchings can be performed in polynomial time, which the constraint solver is able to detect

and hence calculate the matching more efficiently. Finally, the time for the synthesis is marginal even though theoretically complex. This is because the behaviour of each component remains simple while the complexity emerges from the interaction between components.

Figure 6.14 illustrates the time ratio for each mediation step. It can be seen that the ontology encoding is the most time-consuming step. Still, ontologies are static entities since they represent knowledge and understanding of the application domain and are not specific to the components to be made interoperable. Hence, the ontology encoding can be performed beforehand and used when the need arises. The time ratio for interface matching varies greatly between case studies according to the type of matching: whether one-to-one or one-to-many. Finally, the time for generating the mediator based on the computed matchings represents only a small part of the overall processing time and is negligible compared to ontology processing. So, although the use of ontology allows us to reason about the semantics of data and operations and hence increases the level of automation, it comes with a cost. While standards such as OWL-S or WSMO amalgamate data semantics and behaviour and use ontology to represent and reason about both, our approach only uses ontologies when needed and relies on appropriate formalisms for behaviour analysis thereby making the best use of both formalisms.

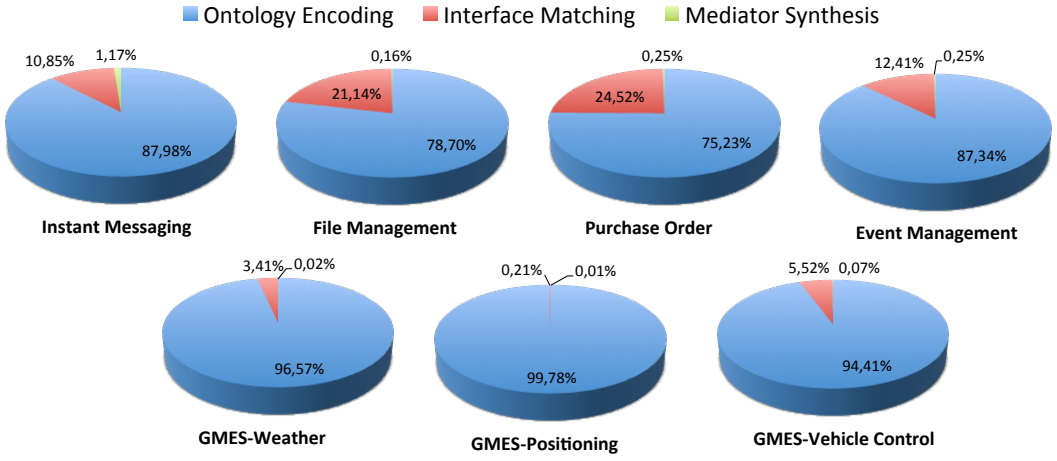

Figure 6.14. Comparison of the time necessary for each mediation step

6.4 Summary

In this chapter, we presented the MICS tool and used it for the synthesis and implementation of mediators in various case studies. The experiments have shown that MICS achieves the main goal of this thesis; that is, automatically enabling interoperability between functionally-compatible components that have different interfaces and interact according to different behaviours at both the application and middleware layers. In addition, the synthesised mediators only introduce a small overhead. More specifically, the different case studies serve to justify the following claims:

- We automatically generate mediators that not only translate one action required by one component into an action provided by the other component, but also sequences of actions between components.
- We manage differences between the components' interfaces and behaviours. While interface matching identifies correspondences between the actions of the components' interfaces, the matching processes specify how the translation is performed. During mediator synthesis, we consider how the matching processes must be composed based on the components' behaviours in order to guarantee that the composed system is deadlock-free.
- We handle differences in the implementations of components at both the application and middleware layers. The main idea is first to synthesise the mediator using knowledge about the application domain, then to refine it by taking into account the characteristics of the middleware solutions underneath.
- We synthesise mediators at runtime without requiring any human intervention. When integrated in the CONNECT architecture, the synthesis of mediators may obtain the models of functionally-compatible components using discovery and learning to locate the components available in the environment dynamically and complete their model automatically. The CONNECT approach enables interoperability to be achieved in a future-proof manner.

Furthermore, although we did not concentrate on the performance of mediators *per se*, we showed that the synthesised mediators introduce only a small overhead.

Chapter 7

Conclusion

> *" To accomplish great things, we must not only act, but also dream; not only plan, but also believe."*
>
> — Anatole France, poet, journalist, and novelist (1844-1924)

At the beginning of this thesis, we asked a number of questions. How can I chat with my friend on Yahoo! using my enterprise messaging service? How can I open my Google Docs files using the Finder application on my Mac? How can a company use the same application to order products from different providers? How can the command and control centre of one country effectively use the resources offered by another country in emergency situations? We subsequently defined an approach to synthesise and implement mediators that enable interoperability in all the above examples, thereby answering these questions. Our aim is to enable users to take advantage of the services surrounding them seamlessly and to let developers focus on creating innovative services rather than on plumbing to enable components to interoperate. But on our way to answering the aforementioned questions, new ones arise opening up perspectives for further research. In this concluding chapter, we summarise our contributions and present future work.

7.1 Contributions

This thesis tackles interoperability between software components in ubiquitous computing environments. We presented an approach to achieve interoperability based on

the automated synthesis and implementation of mediators. Our contribution beyond the state-of-the-art primarily lies in handling interoperability from the application to the middleware layer in an integrated way. As depicted again in Figure 7.1, the mediators we synthesise act as: (i) translators by ensuring the meaningful exchange of information between components, (ii) controllers by coordinating the behaviours of the components to ensure the absence of errors in their interaction, and (iii) middleware by enabling the interaction of components across the network so that each component receives the data it expects at the right moment and in the right format.

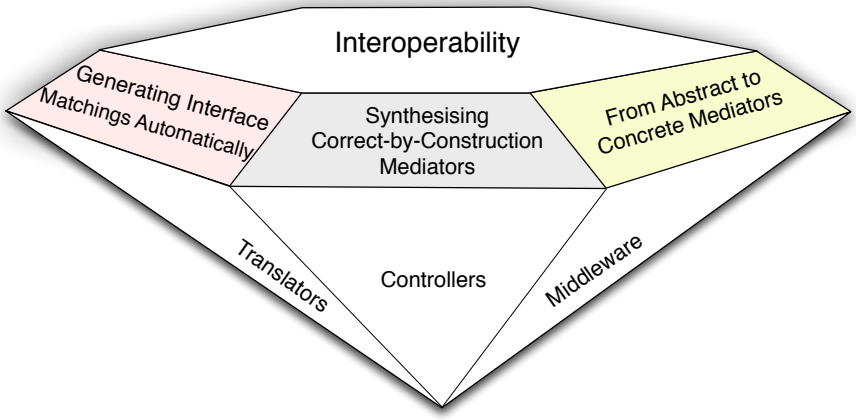

Figure 7.1. A multifaceted approach for interoperability

The synthesis and implementation of mediators is performed in several steps. The first step is interface matching, which identifies the semantic correspondence between the actions required by one component and those provided by the other. We incorporated the use of ontology reasoning within constraint solvers, by defining an encoding of the ontology relations using arithmetic operators supported by widespread solvers, and use it to perform interface matching efficiently. For each identified correspondence, we generate an associated matching process that performs the necessary translations between the actions of the two components' interfaces. The second step is the synthesis of correct-by-construction mediators. To do so, we analyse the behaviours of components so as to generate the mediator that combines the matching processes in a way that guarantees that the two components progress and reach their final states without errors. The last step consists in making the synthesised mediator concrete by incorporating all the details about the interaction of components. To do so, we compute the translation functions necessary to reconcile the differences in the

syntax of the input/output data used by each component and coordinate the different interaction patterns that can be used by middleware solutions.

We developed the MICS tool to carry out the automated synthesis and implementation of mediators and experimented it with a number of case studies that demonstrated that the synthesised mediators enable components to interoperate while introducing only a small overhead. The systematic approach for mediator synthesis lays firm foundations for dealing with interoperability in an increasingly heterogeneous world.

7.2 Future Work

Automated mediator synthesis and implementation offers opportunities both in the short term for investigating extensions and enhancements to the approach and the MICS tool, and in the long term for exploring new research directions.

7.2.1 Mediator Synthesis as a Service

In this thesis, we identified the several roles a mediator must play: translator, controller, and middleware. We then defined a multifaceted approach that deals with them all: automated generation of interface matchings to realise translation, computation of correct-by-construction mediators to deal with control, and implementation of the mediator to deal with middleware concerns. Our approach handles all the steps without human intervention. Nonetheless, we require rich models of the components and extract matching for which automated translation can be generated. Another alternative would be to accept additional transformations by the developers. This raises the issue of the validity of user-specified matchings but could enable a more flexible way to perform the synthesis. In fact, we can augment the matching computation with an interactive and graphical environment, such as that proposed by ITACA [CMS+09], to allow developers to specify additional matchings.

It is our belief that the formal definition of mediator synthesis, as we presented it, allows us to define a mediator synthesis service where each step of mediation can be performed using different methods while the appropriate method is selected according to the context, as depicted in Figure 7.2. Indeed, as described in Chapter 2, many approaches have been proposed to perform one mediation step. Most of the

time, these approaches differ in their requirements; for example, some require user intervention either to define interface matching or goals. Such a service would also allow us to compare existing approaches from both a qualitative and a quantitative perspective.

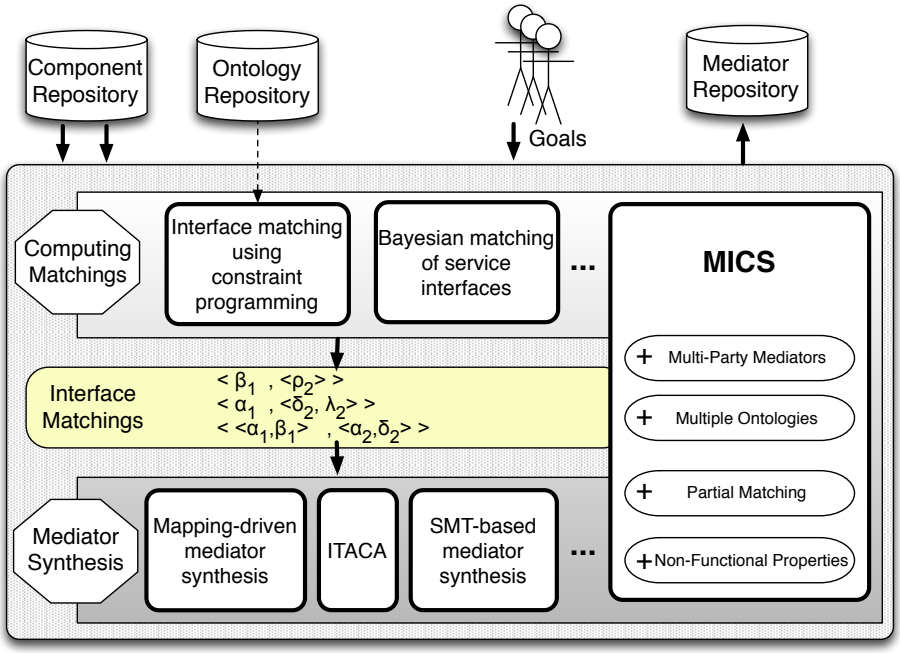

Figure 7.2. Mediator Synthesis as a Service

Furthermore, we can build upon existing solutions and extend the work of this thesis, and MICS, in many directions as discussed below.

Multi-Party Mediators

With the advent of social-based interactions and the increased emphasis on collaboration, interoperability between multiple —more than two— components is gaining momentum. One simple solution to handle this case is by combining assembly methods (e.g., [SMK11]) with pairwise mediators. The former consider the structural constraints and specify a coarse-grained composition of components based on their capabilities, while the latter take care of enforcing this composition despite the interface and behavioural differences that may exist between each pair of components.

Dealing with Multiple Ontologies

It is crucial to think about ontologies as a means to interoperability rather than universality. Hence, it is often the case that many ontologies co-exist and need to be matched with one another. Ontology matching techniques primarily exploit knowledge explicitly encoded in the ontology rather than trying to guess the meaning encoded in the schemas, as is the case with XML schemas, for example. More specifically, while XML schema matching techniques rely on the use of statistical measures of syntactic similarity, ontologies deal with axioms and how they can be put together [SE05]. This can further benefit from the existence of upper ontologies such as DOLCE [GGM+02] and SUMO [NP01], which are domain-independent ontologies. Still, these matching techniques may induce some inaccuracy that the synthesis algorithm must deal with. In the future, we aim to extend our model so as to consider the heterogeneity of the ontologies themselves and reason about interface matching based on imprecise information.

Partial Matching

In our approach, we postulate that all actions required by one component must match with an action or sequence of actions provided by the other component. This requirement allows us to prove that the mediated system is free from deadlocks. Nevertheless, it is interesting to authorise partial correspondence between actions so long as the task required by users or developers can be achieved. Indeed, the user may be interested in achieving only one specific task and we can permit interaction between components if we can mediate their behaviours in order to perform this specific task. Therefore, we can use projections [LS84] to retain only the components' behaviours related to this task and then perform the mediation.

Dealing with Non-Functional Properties

In this thesis, we focused on the functional characteristics of components represented by their capabilities, interfaces, and behaviours. Non-functional properties allow us to represent how well the component achieves its functionality. There are some non-functional properties that are very similar to functional ones and can be represented using the same formalisms. For example, the work of Spitznagel and Garlan [SG03] specifically targets dependability using FSP and connector transformations. However,

some other non-functional properties, mainly quantitative ones, require changing the formalism to be able to perform automated reasoning.

7.2.2 Mediator Evolution

The vision of eternal interoperability we presented in Chapter 3 claims that any solution to interoperability may turn out to be inappropriate over time. Hence, the mediator generated at a given time may become unsuitable as time passes or the environment changes. The idea is to make the mediator evolve. Closed-loop systems have been recognised as fundamental for dealing with change [BDNG06, Gar10]. Hence, the architecture of enablers supporting the creation of mediators can also be used to make the mediator evolve, as depicted in Figure 7.3.

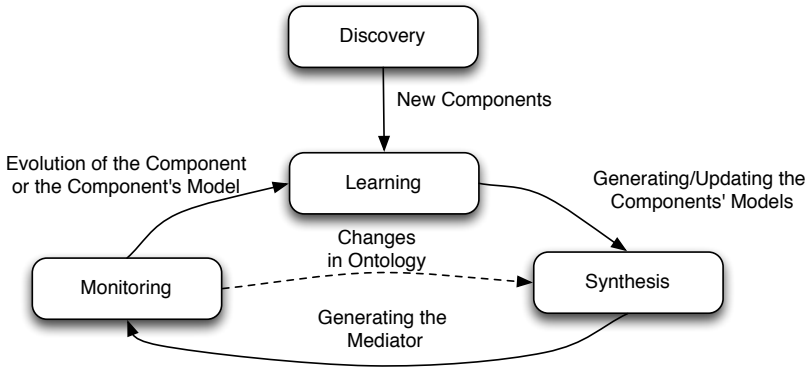

Figure 7.3. Towards mediator evolution

The main threat to the validity of the mediator is the inaccuracy in the models of the components. Indeed, the correctness of the mediator is conditioned by the correctness of the behaviours of the components for which it was synthesised. While machine learning significantly improves automation by completing the model of the component based on its interface, it also induces some inaccuracy that may lead the system to reach an erroneous state. This inaccuracy is inherent in learning techniques and cannot totally be removed. Hence, we should accept this imprecision and apply engineering techniques that are intended to increase precision over time, such as a control loop in which the system is continuously monitored so as to evaluate the correspondence between the actual component and its model. Mediator evolution aims at preserving/re-establishing a mediator that becomes invalid because of some uncontrolled changes either in the components or their models.

When the model of a component changes due to behavioural learning, the synthesis of the mediator can be based on the previous interface matching and ontology encoding, which are the most time-consuming phases. When one of the components changes its interface, then the synthesis has to resume from interface matching. Finally, if the ontology evolves, the synthesis has to be restarted from the ontology encoding. In this context, incremental re-synthesis would be very important to make the mediator evolve at a low cost.

Inaccuracy may also be due to imprecision in the ontology, especially if it is built by merging several ontologies. In this case, not only the mediator has to evolve, but so must the ontology in order to reflect better understanding of the domain. Hence, the automated synthesis and implementation of mediators open new research perspectives yet to be explored.

7.3 One More Thing...

At school we were used to having teachers giving us rules (let us say for addition) and asking us to apply them on several examples. Moving up to higher education, the teachers started giving us more examples, which we had to investigate in order to find the rules for ourselves. However, when doing a PhD, neither the examples nor the rules are available. We have to define the problem, find representative examples and infer the rules ourselves. Nevertheless, while we may be able to apply or extract rules on our own, defining a problem for a PhD can only be done through discussion and exchange with other people. Why am I telling you that? Let us go back to computers. At the beginning, software development was simply giving rules or instructions to computers in the form of programs; the computer executes the rules and gives us the correct results. Nowadays, there is an increasing emphasis on learning as a means to infer programs from examples. Enabling computers to learn is fast becoming reality with some significant success stories, e.g., IBM Watson. But the computers of the future should be able to observe the world, find their own examples and infer the appropriate rules. They should be able to innovate, as PhD students seek to do. Of course, this can only be done if computers can interact and share knowledge not only with humans, as is the case with ontologies, but also with other computers. While major research challenges remain, we hope that this thesis demonstrated the potential of the automated synthesis and implementation of mediators to contribute to this vision, by enabling the seamless interaction of computers.

Appendix A

FSP Syntax & Semantics

In this appendix we succinctly define the syntax and semantics of the FSP process algebra. We refer the interested reader to [MK06] for a complete reference.

Definitions	
αP	The alphabet of a process P
END	Predefined process, denotes the state in which a process successfully terminates
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P(X =' a)$	Parameterised process: P is described using parameter X and modelled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ..., new\_n/old\_n\}$	Relabelling

Composite Processes ($\|P$)	
$P\|\|Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $(P(1)\|\|...\|\|P(n))$.
$a : P$	Process labelling

FSP processes describe actions (events) that occur in sequence, and choices between event sequences. Each process has an alphabet of the events that it is aware of (and either engages in or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix, choice, and sequential composition. Composite processes are constructed using parallel composition or process relabelling. When composed in parallel, processes synchronise on shared events: if processes P and Q are composed in parallel as $P||Q$, events that are in the alphabet of only one of the two processes can occur independently of the other process, but an event that is in the alphabets of both processes cannot occur until the two of them are willing to engage in it. The replicator forall is a convenient syntactic construct used to specify parallel composition over a set of processes. Processes can optionally be parameterised and have relabelling, hiding or extension over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with $\|$.

The semantics of FSP is defined in terms of Labelled Transition Systems (LTS). The LTS associated with an FSP process P is a quadruple $lts(P) = \langle S, A, \Delta, s_0 \rangle$ where:

- S is a finite set of states,
- $A = \alpha P \cup \tau$ represents the alphabet of the LTS. τ is used to denote an internal action that cannot be observed by the environment of an LTS,
- $\Delta \subseteq S \times A \times S$ denotes a transition relation that specifies that if the process is in state $s \in S$ and engages in an action a, then it transits to state s', written $s \xrightarrow{a} s'$. $s \stackrel{s}{\Rightarrow} s', X = \langle a_1, ..., a_n \rangle$, $a_i \in \mathcal{I}$ is a shorthand for $s \xrightarrow{a_1} s_1 ... \xrightarrow{a_n} s'$, and
- $s_0 \in S$ indicates the initial state.

In addition, an LTS terminates if there is a state $e \in S$ such that $\nexists(e, a, s_0) \in \Delta$.

Let $lts(P) = \langle S_P, A_P, \Delta_P, s_{0P} \rangle$ and $lts(Q) = \langle S_Q, A_Q, \Delta_Q, s_{0Q} \rangle$ be the LTSs associated with P and Q respectively. The semantics of FSP is as follows.

FSP Semantics	
END	$lts(\mathsf{END}) =< \{e\}, \{\tau\}, \{\}, e >$ where e is a terminating state
$a \rightarrow P$	$lts(a \rightarrow P) =< S_P \cup \{s_n\}, A_P \cup \{a\}, \Delta_P \cup \{(s_n, a, s_{0P})\}, s_n >$ where $s_n \notin S_P$
$a \rightarrow P \vert b \rightarrow Q$	$lts(a \rightarrow P \vert b \rightarrow Q) =< S_P \cup S_Q \cup \{s_n\}, A_P \cup A_Q \cup \{a, b\}, \Delta_P \cup \Delta_Q \cup \{(s_n, a, s_{0P}), (s_n, b, s_{0Q})\}, s_n >$ where $s_n \notin S_P$ and $s_n \notin S_Q$
$P; Q$	$lts(P; Q) =< S_P \cup S_Q, A_P \cup A_Q, \Delta_P \cup \Delta_Q, s_{0P} >$ where $s_{0Q} = e_p$ and e_p is terminating state of $lts(P)$
$P \Vert Q$	$lts(P \Vert Q) =< S_P \times S_Q, A_P \cup A_Q, \Delta, (s_{0P}, s_{0Q}) >$ where Δ is the smallest relation satisfying the rules $\frac{P \xrightarrow{a} P', \nexists a \in \alpha Q}{P \Vert Q \xrightarrow{a} P' \Vert Q}$ $\frac{Q \xrightarrow{a} Q', \nexists a \in \alpha P}{P \Vert Q \xrightarrow{a} P \Vert Q'}$ $\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q', a \neq \tau}{P \Vert Q \xrightarrow{a} P' \Vert Q'}$

Appendix B

DL Syntax & Semantics

In this appendix we introduce the basic DL constructs used in the paper. In particular, we define how composite concepts can be built from them inductively with concept constructors and role constructors. We refer the interested reader to [BCM+03] for a complete reference.

An interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept C a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic object property R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. C and D are concepts and R is an object property. The interpretation is extended to concept description following inductive definitions.

	Syntax	**Semantics**
Atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Top built-in concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Bottom built-in concept	$\bot$	$\bot^{\mathcal{I}} = \emptyset$
Complement	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Universal quantifier	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
Existential quantifier	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Aggregation	$C \oplus D$	$(C \oplus D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in \mathsf{hasPart}^{\mathcal{I}} \wedge y \in C^{\mathcal{I}} \wedge \exists z.(x,z) \in \mathsf{hasPart}^{\mathcal{I}} \wedge z \in D^{\mathcal{I}}\}$

References

[ABL+07] M. E. Aranguren, S. Bechhofer, P. W. Lord, U. Sattler, and R. D. Stevens. Understanding and using the meaning of statements in a bio-ontology: recasting the gene ontology in OWL. *BMC bioinformatics*, 8(1):57, 2007. 44

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions Software Engineering Methodology*, 6(3):213–249, 1997. 22, 23, 26, 51

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. 63

[BBC05] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005. 12

[BBG+11] 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 *Proc. of Middleware*, pages 410–430, 2011. 15

[BBG+13] N. Bencomo, A. Bennaceur, P. Grace, G. S. Blair, and V. Issarny. The role of models@run.time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190, 2013. 15

[BCI+13] A. Bennaceur, C. Chilton, M. Isberner, , and B. Jonsson. Automated mediator synthesis: Combining behavioural and ontological reasoning. In *Proc. of the 11th IEEE International Conference on Software Engineering and Formal Methods, SEFM*, 2013. to appear. 15

[BCK12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Pearson Education, 2012. 3

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003. 44, 45, 157

[BCV+08] M. Brambilla, S. Ceri, E. Valle, F. Facca, and C. Tziviskou. A software engineering approach based on WebML and BPMN to the mediation scenario of the sws challenge. In *Semantic Web Services Challenge: Results from the First Year*, pages 51–70. Springer, 2008. 133, 135

[BDNG06] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006. 150

[BGR11] Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: runtime interoperability between heterogeneous middleware protocols. In *International Conference on Distributed Computing Systems, ICDCS*, pages 446–455, 2011. 33, 51, 58, 59

[BHL+02] M. H. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. Daml-s: Web service description for the semantic web. In *International Semantic Web Conference, ISWC*, pages 348–363, 2002. 46, 51

[BI13] A. Bennaceur and V. Issarny. Automated synthesis of mediators to support component interoperability. *IEEE Transactions on Software Engineering*, 2013. Submitted. 15

[BIR+11] A. Bennaceur, V. Issarny, J. Richard, M. Alessandro, S. Romina, and D. Sykes. Automatic service categorisation through machine learning in emergent middleware. In *Software Technologies Concertation on Formal Methods for Components and Objects, FMCO*, pages 133–149, 2011. 15

[BIS+12] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti. Machine learning for emergent

middleware. In *Proc. of the Joint workshop on Intelligent Methods for Software System Engineering, JIMSE*, 2012. 15

[BIST12] A. Bennaceur, V. Issarny, R. Spalazzese, and S. Tyagi. Achieving interoperability through semantics-based technologies: The instant messaging case. In *11th International Semantic Web Conference, ISWC*, pages 17–33, 2012. 15

[BLHL+01] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001. 4, 12, 44

[BN84] A. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions Computing System*, 2(1):39–59, 1984. 105

[BOR04] S. Becker, S. Overhage, and R. Reussner. Classifying software component interoperability errors to support component adaption. In *Proc. of the 7th International Symposium on Component-Based Software Engineering, CBSE*, pages 68–83, 2004. 19

[BPGG11] G. Blair, M. Paolucci, P. Grace, and N. Georgantas. Interoperability in complex distributed systems. In M. Bernardo and V. Issarny, editors, *SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems*, pages 1–26. Springer Verlag, 2011. 31

[BPT10] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010. 39, 51

[BRA+11] A. Bennaceur, J. Richard, M. Alessandro, S. Romina, D. Sykes, R. Saadi, and V. Issarny. Inferring affordances using learning techniques. In *International Workshop on Eternal Systems, EternalS*, pages 79–87, 2011. 15

[BRLM09] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Proc. of Middleware*, 2009. 33, 51

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. 59

[BZ01] N. Busi and G. Zavattaro. Publish/subscribe vs. shared dataspace coordination infrastructures: Is it just a matter of taste? In *Proc. of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, pages 328–333, 2001. 116

[CDKB12] G. F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed systems: concepts and design, Fifth Edition.* Addison-Wesley Longman, 2012. 104

[CK10] R. Calinescu and S. Kikuchi. Formal methods @ runtime. In *16th Monterey Workshop on Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, pages 122–135, 2010. 12

[CL89] K. L. Calvert and S. S. Lam. Deriving a protocol converter: A top-down method. In *Proc. of the Symposium on Communications Architectures & Protocols, SIGCOMM*, pages 247–258, 1989. 38, 51

[CM05] E. Cimpian and A. Mocan. WSMX process mediation based on choreographies. In *Proc. of Business Process Management Workshop*, pages 130–143, 2005. 48, 51

[CMP08] M. Ceriotti, A. L. Murphy, and G. P. Picco. Data sharing vs. message passing: synergy or incompatibility?: an implementation-driven case study. In *Proc. of the ACM Symposium on Applied Computing, SAC*, pages 100–107, 2008. 11, 36, 116

[CMP09] H. Chang, L. Mariani, and M. Pezzè. In-field healing of integration problems with COTS components. In *International Conference on Software Engineering, ICSE*, pages 166–176, 2009. 29, 51

[CMS+09] J. Cámara, J. A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An integrated toolbox for the automatic com-

position and adaptation of web services. In *Proc. of the International Conference on Software Engineering, ICSE*, pages 627–630, 2009. 147

[Con12] C. Consortium. CONNECT Deliverable D6.4: Assessment report: Experimenting with CONNECT in Systems of Systems, and Mobile Environments. FET IP CONNECT EU project., 2012. 8, 57, 140

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. 11

[DCtTdK11] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web*, 2(2):71–87, 2011. 45

[DG09] J. Davies and J. Gibbons. Formal methods for future interoperability. *SIGCSE Bulletin*, 41(2):60–64, 2009. 12

[dN12] M. d'Aquin and N. F. Noy. Where to publish and find ontologies? a survey of ontology libraries. *J. Web Sem.*, 11:96–111, 2012. 44

[DPT09] G. Denaro, M. Pezzè, and D. Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 253–262, 2009. 29, 51

[Fos08] H. Foster. WS-Engineer 2008. In *Proc. of the 6th International Conference on Service-Oriented Computing, ICSOC*, pages 728–729, 2008. 22

[Fre91] E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. of the 9th National Conference on Artificial Intelligence, AAAI*, pages 227–233, 1991. 82

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proc. of the 17th International Conference on Software Engineering, ICSE*, pages 179–185, 1995. 21, 51

[Gar10] D. Garlan. Software engineering in an uncertain world. In *Proc. of the Workshop on Future of Software Engineering Research, FoSER*, pages 125–128, 2010. 150

[GBS03] P. Grace, G. S. Blair, and S. Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *Proc. of the OTM Confederated International Conferences CoopIS/DOA/ODBASE*, pages 1170–1187, 2003. 32, 51

[GGM+02] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening ontologies with DOLCE. In *Proc. of the 13th International Conference on Knowledge Engineering and Knowledge Management, EKAW*, pages 166–181, 2002. 149

[GIBM+10] N. Georgantas, V. Issarny, S. Ben Mokhtar, Y.-D. Bromberg, S. Bianco, G. Thomson, P.-G. Raverdy, A. Urbieta, and R. S. Cardoso. Middleware architecture for ambient intelligence in the networked home. In H. Nakashima, H. Aghajan, and J. Augusto, editors, *Handbook of Ambient Intelligence and Smart Environments*, pages 1139–1169. Springer, 2010. 34, 51

[GMW12] C. Gierds, A. J. Mooij, and K. Wolf. Reducing adapter synthesis to controller synthesis. *IEEE Transactions on Services Computing*, 5(1):72–85, 2012. 39, 51

[GR08] J. Golbeck and M. Rothstein. Linking social networks on the web with foaf: A semantic web case study. In *Proc. of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1138–1143, 2008. 44

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993. 12

[Gru09] T. Gruber. Ontology. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1963–1965. Springer US, 2009. 4, 44

[GRW+08] K. Gomadam, A. Ranabahu, Z. Wu, A. Sheth, and J. Miller. A declarative approach using SAWSDL and semantic templates towards process

mediation. In *Semantic Web Services Challenge: Results from the First Year*, pages 101–118. Springer, 2008. 134, 135

[Gua04] N. Guarino. Helping people (and machines) understanding each other: The role of formal ontology. In *CoopIS/DOA/ODBASE (1)*, page 599, 2004. 44

[Hoa04] C. A. R. Hoare. Process algebra: A unifying approach. In *25 Years Communicating Sequential Processes*, 2004. 26

[IB13] V. Issarny and A. Bennaceur. Composing distributed systems: Overcoming the interoperability challenge. In *HATS International School on Formal Models for Components and Objects, HATS-FMCO*. Springer Verlag, 2013. to appear. 15

[IBB11] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In M. Bernardo and V. Issarny, editors, *SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems*, pages 217–255. Springer Verlag, 2011. 15

[ICG07] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. In *Proc. of the Workshop on the Future of Software Engineering, FOSE*, pages 244–258, 2007. 20, 104

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Standard*, 610121990:121990, 1990. 3

[ISO11] ISO/IEC TR 29110-1. Software engineering–lifecycle profiles for very small entities (vses)–part 1: Overview, 2.33. Technical report, ISO, 2011. 4

[IT13] 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*, pages 3–12, 2013. 28, 51

[JGGA+13] R. Jardim-Gonçalves, A. Grilo, C. Agostinho, F. Lampathaki, and Y. Charalabidis. Systematisation of interoperability body of knowledge: the foundation for enterprise interoperability as a science. *Enterprise IS*, 7(1):7–32, 2013. 2

[JM03] U. Junker and D. Mailharro. The logic of ilog (j) configurator: Combining constraint programming with a description logic. In *Proc. of IJCAI Workshop on Configuration*, volume 3, pages 13–20, 2003. 79

[Joa98] T. Joachims. Text categorization with suport vector machines: Learning with many relevant features. In *Proc. of the 10th European Conference on Machine Learning, ECML*, pages 137–142, 1998. 62

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Proc. of a Symposium on the Complexity of Computer Computations*, pages 85–103, 1972. 80

[KBP+10] 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)*, pages 179–182, 2010. 63

[KC09] H. Kubicek and R. Cimander. Three dimensions of organizational interoperability. *European Journal of ePractice*, 6, 2009. 2

[Kel76] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. 23

[KMSN08] C. Kubczak, T. Margaria, B. Steffen, and R. Nagel. Service-oriented mediation with jABC/jETI. In *Semantic Web Services Challenge: Results from the First Year*, pages 71–99. Springer, 2008. 134, 135

[KVBF07] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6):60–67, 2007. 49

[Lab03] F. Laburthe. Constraints over ontologies. In *Proc. of the 9th International Conference on Principles and Practice of Constraint Programming,CP*, pages 878–882, 2003. 80

[Lam88] S. S. Lam. Protocol conversion. *IEEE Transaction Software Engineering*, 14(3):353–362, 1988. 40, 51

[Lar03] E. Larson. Interoperability of us and nato allied air forces: Supporting data and case studies. Technical Report 1603, RAND Corporation, 2003. 5

[LMP08] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. of the International Conference on Software Engineering, ICSE*, pages 501–510, 2008. 63

[LMSW08] G. A. Lewis, E. J. Morris, S. Simanta, and L. Wrage. Why standards are not enough to guarantee end-to-end interoperability. In *Seventh International Conference on Composition-Based Software Systems, ICCBSS*, pages 164–173, 2008. 4

[LS84] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions Software Engineering*, 10(4):325–342, 1984. 149

[LW94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 16(6):1811–1841, 1994. 74

[MBM+07] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with OWL-S. In *Proc. if the World Wide Web conference, WWW'07*, pages 243–277, 2007. 46, 51

[MDT03] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. The role of middleware in architecture-based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003. 20, 28, 30

[Men07] F. Menge. Enterprise Service Bus. In *Proc. of the Free and open source Software conf.*, 2007. 35, 51

[Mic10] Microsoft Press. Yahoo! and microsoft bridge global instant messaging communities, 2010. online. 129

[MK06] J. Magee and J. Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006. 22, 23, 153

[MLM+04] E. Morris, L. Levine, C. Meyers, P. Place, and D. Plakosh. System of systems interoperability (sosi): final report. Technical report, Software Engineering Institute, Carnegie Mellon University, 2004. 5

[MPR06] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions Software Engineering Methodology*, 15(3), 2006. 107

[MPS12] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Transactions Software Engineering*, 38(4):755–777, 2012. 12, 42, 51

[MSHM11] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *Proc. of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 220–223, 2011. 63

[MSZ01] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001. 12

[NBCT06] H. R. M. Nezhad, B. Benatallah, F. Casati, and F. Toumani. Web services interoperability specifications. *IEEE Computer*, 39(5):24–32, 2006. 6, 36

[NBM+07] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proc. of the 16th International Conference on World Wide Web, WWW*, pages 993–1002, 2007. 42

[Nie93] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. 129

[NP01] I. Niles and A. Pease. Towards a standard upper ontology. In *Proc. of the 2nd International Conference on Formal Ontology in Information Systems, FOIS*, pages 2–9, 2001. 149

[NXB10] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proc. of the 19th International Conference on World Wide Web, WWW*, 2010. 42, 51

[PKPS02] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *Proc. of the First International Semantic Web Conference, ISWC*, 2002. 47

[PMLZ08] C. Petrie, T. Margaria, H. Lausen, and M. Zaremba. *Semantic Web Services Challenge: Results from the First Year*, volume 8. Springer, 2008. 8, 124, 131, 133

[PSDZ12] D. Papadimitriou, B. Sales, P. Demeester, and T. Zahariadis. From internet architecture research to standards. In *Future Internet Assembly*, pages 68–80, 2012. 4

[Rad12] Radicati Group. *Instant Messaging Market 12-16*, 2012. 7

[RP05] R. G. Raskin and M. J. Pan. Knowledge representation in the semantic web for earth and environmental terminology (SWEET). *Computers & Geosciences*, 31(9):1119–1125, 2005. 44

[RVBW06] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*, volume 35. Elsevier Science, 2006. 79, 81

[Sal10] G. Salaün. Analysis and verification of service interaction protocols - a brief survey. In *Proc. of the 4th International Workshop on Testing, Analysis and Verification of Web Software, TAV-WEB*, pages 75–86, 2010. 12

[SE05] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal of Data Semantics IV*, pages 146–171, 2005. 99, 149

[SEC12] C. SECUR-ED. Deliverable d22.1: Interoperability concept. fp7 SECUR-ED EU project., 2012. 5

[SG03] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proc. of the 25th International Conference on Software Engineering, ICSE*, pages 374–384, 2003. 22, 28, 51, 149

[Sha93] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE Workshop on Studies of Software Design*, pages 17–32, 1993. 9, 19

[SMH+10] L. Seligman, P. Mork, A. Y. Halevy, K. P. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. Openii: an open source information integration toolkit. In *Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 1057–1060, 2010. 99

[SMK11] D. Sykes, J. Magee, and J. Kramer. Flashmob: distributed adaptive self-assembly. In *Proc. of the ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, pages 100–109, 2011. 148

[SYY74] G. Salton, C. S. Yang, and C. T. Yu. Contribution to the theory of indexing. In *IFIP Congress*, pages 584–590, 1974. 62

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice.* Hoboken (N.J.) : Wiley, 2009. 4, 19, 20

[Tol03] A. Tolk. Beyond technical interoperability - introducing a reference model for measures of merit for coalition interoperability. In *Proc. of the International Comamand and Control Research and Technology Symposium*, 2003. 2

[TVS06] A. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms - Second Edition.* Prentice Hall, 2006. 3, 5, 10, 104

[TW10] A. S. Tanenbaum and D. J. Wetherall. *Computer networks (5th edition).* Prentice Hall, 2010. 4

[UCJ09] J. Ullberg, D. Chen, and P. Johnson. Barriers to enterprise interoperability. In *Proc. of the 2nd International Workshop on Enterprise Interoperability, IWEI*, pages 13–24, 2009. 19

[VHPK04] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In *Proc. of the 9th International Conference on Reliable Software Technologies Reliable Software Technologies, Ada-Europe*, pages 106–119, 2004. 32, 51

[VNS09] R. Vaculín, R. Neruda, and K. P. Sycara. The process mediation framework for semantic web services. *International Journal of Agent-Oriented Software Engineering, IJAOSE*, 3(1):27–58, 2009. 47

[VZMM08] T. Vitvar, M. Zaremba, M. Moran, and A. Mocan. Mediation using WSMO, WSML and WSMX. In *Semantic Web Services Challenge: Results from the First Year*, pages 31–49. Springer, 2008. 133, 135

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992. 5

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and System, TOPLAS*, 19(2):292–333, 1997. 12, 41, 51

[Zhu12] W. Zhu. Semantic mediation bus: An ontology-based runtime infrastructure for service interoperability. In *Proc. of the 16th International on Enterprise Distributed Object Computing Conference Workshops, EDOCW*, pages 140 –145, sept. 2012. 49, 51