UNIVERSITY OF SOUTHAMPTON

Dynamic Discovery, Creation and Invocation of Type Adaptors for Web Service Workflow Harmonisation

by

Martin Szomszor

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

April 2007

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Martin Szomszor

Service-oriented architectures have evolved to support the composition and utilisation of heterogeneous resources, such as services and data repositories, whose deployments can span both physical and organisational boundaries. The Semantic Web Service paradigm facilitates the construction of workflows over such resources using annotations that express the meaning of the service through a shared conceptualisation. While this aids non expert users in the composition of meaningful workflows, sophisticated middleware is required to cater for the fact that service providers and consumers often assume different data formats for conceptually equivalent information. When syntactic mismatches occur, some form of *workflow harmonisation* is required to ensure that data incompatibilities are resolved, a step we refer to as *syntactic mediation.* Current solutions are entirely manual; users must consider the low-level interoperability issues and insert *Type Adaptor* components into the workflow by hand, contradicting the Semantic Web Service ideology.

By exploiting the fact that services are connected together based on shared conceptual interfaces, it is possible to associate a canonical data model with these shared concepts, providing the basis for workflow harmonisation through this intermediary data model. To investigate this hypothesis, we have developed a formalism to express the mapping of elements between data models in a modular and composable fashion. To utilise such a formalism, we propose additional architecture that facilitates the discovery of declarative mediation rules and subsequent on-the-fly construction of Type Adaptors that can translate data between different syntactic representations. This formalism and proposed architecture have been implemented and evaluated against bioinformatics data sources to demonstrate a scalable and efficient solution that offers composability with virtually no overhead. This novel mediation approach scales well as the number of compatible data formats increases, promotes the sharing and reuse of mediation rules, and facilitates the automatic inclusion of Type Adaptor components into workflows.

Contents

List of Figures

Acknowledgements

I would like to thank my two supervisors, Luc Moreau and Terry R. Payne, for their continued support, encouragement and patience. In particular, I thank them for helping me develop the academic skills and personal fortitude required to complete this Thesis. I would also like to thanks my colleagues at the University of Southampton for their stimulating conversation and motivation, particularly Paul Groth, Mischa Tuffield, Seb Francois, Antonis Loizou, Maria Karam, Roxana Belecheanu, Steve Munroe, Danius Michaelides, Ayomi Bandra, and Hugh Glaser.

Outside of the University, I extend my biggest thanks to my family. Over the past three years they have supplied me with an incentive to work hard, given me valuable advice, and provided a refuge in my times of need.

Finally, I give a special mention to Laura. Her vitality and affection is a constant inspiration.

Chapter 1

Introduction

During the latter half of the 20^{th} Century, scientists took the initiative to build a global communication medium to support the transmission of information between parties located anywhere on the planet. Their efforts culminated in the 1990s with the appearance of what is now commonly recognised as the Internet: a world-wide network of interconnected computers supporting the reliable interchange of data. The Internet itself should not be considered as a monolithic entity but rather a dynamic and loosely coupled collection of smaller networks managed by businesses, academic institutions, governments, and individuals, all sharing a diverse range of information exposed in a rich variety of formats.

With an explosion in the volume and connectivity of computing resources, the requirements to manage computations across large, geographically separated, heterogeneous resources have become more complex. Information can be spread across different storage end-points in a variety of different formats, each with different access models. Grid [44] and Web Services [23] have evolved to support applications operating in these types of environment, enabling the collation of computing assets to meet complex computing requirements through the use of *service-oriented architectures* (SOAs). SOAs are founded on a perspective that facilitates the consolidation of loosely coupled, dynamic resources, by adhering to a uniform access model that hides the underlying implementation. This facilitates cost effective and rapid adaptation to changes in requirements, and the convenient incorporation of new resources, while maintaining high levels of interoperability. By providing uniform access to resources spanning both physical and organisational boundaries, SOAs allow users to gather information from disparate resources, perform intensive computational analysis, and collect results in sophisticated formats. One application domain that profits from the benefits of such an architecture is eScience where bioinformatics [80], high energy physics [50] and astronomy [11] applications have been developed to assist users in scientific experimentation.

Much of the success of these applications comes from the ability to provide end-users with simple paradigms onto which they can map conventional scientific practices. One key example of this is *workflow*: the specification of a computational process across multiple resources. This is very similar to the design and execution of a scientific experiment which is usually expressed as a workflow with a number of tasks. With SOAs, these scientific tasks are realised by services, allowing users simply to convert their intended experiment directly to a workflow specification. To this end, Grid and Web Services communities strive to provide users with the most productive conditions, supporting them in the discovery of services to meet their goals, and the specification of meaningful workflows.

Recent advances within the Grid and Web Services community have focused on helping users in the discovery of services and their composition to form functioning workflows. As the number of service instances continues to increase, the need for efficient and user-friendly service matching is more important; searching over service descriptions alone is a cumbersome and tedious task. In many cases, service operations are not documented and operation names have little *semantic* value; colloquial terms, acronyms and shorthand concatenations frustrate users and impede the discovery process. However, by utilising Semantic Web [20] approaches, such as the annotation of service descriptions with concepts from an ontology that capture the meaning of Web Services, users can formulate and execute queries using domain specific terminology from a shared conceptualisation, rather than conventional keyword matching. With suitably rich ontologies, users can find the services they need easily, quickly and reliably. This has been realised through the development of ontology languages, such as OWL (the Web Ontology Language) [83], that supports the publishing and sharing of conceptual models on the Web.

With the introduction of semantically-annotated Web Services, workflow composition has shifted to a higher-level design process: users can choose to include services in workflow to achieve particular goals based on a high-level definition of the service capability. While tools [89, 48] that exploit these semantic definitions can make workflow design more accessible to untrained users, it does lead to more complex architectural requirements. The situation often arises where users wish to connect two services together that are *conceptually compatible* but have different *syntactic interfaces*. This occurs when service providers use their own data formats to represent information within their application domain. To reconcile any data incompatibilities in a workflow, *syntactic mediation* is required, often taking the form of a translation script, bespoke application code, or mediation Web Service. Currently, these *Type Adaptor* components must be discovered manually and inserted into the workflow by hand, imposing additional effort on the user [58]. Consequently, they are distracted from the workflow design, spending additional time understanding why an incompatibility has been encountered and how it can be resolved.

To improve on the manual selection and insertion of Type Adaptors, existing Web Service architectures can be augmented to identify when syntactic mediation is required within a workflow, what components are available to carry it out, and how they can be invoked. Semantic Web Service research has addressed this issue to a certain degree [2, 84]: semantic annotations that describe the service capability can be used to give meaning to the information it consumes and produces. By extending existing semantic service definitions to capture the structure and semantics of the data consumed and produced, an ontology can be used as a shared conceptual reference model, facilitating the translation of data between different syntactic representations.

By combining work from the data integration field, Semantic Web research and Web Service invocation techniques, we show that it is possible to supply an architecture that supports automated workflow harmonisation through the automatic discovery and invocation of appropriate Type Adaptors. By investigating a bioinformatics use case, we deduce the requirements for syntactic mediation and the kinds of complex data translation required. Much of our architecture is centred on the development and utilisation of a bespoke mapping language to express the relationship between concrete data formats and their corresponding conceptual models. We derive the requirements for this mapping language from bioinformatics data sets and present a formalism that describes such mappings and the transformation process derived from them.

1.1 Thesis Statement and Contributions

The following thesis statement summarises our solution to the problem of workflow harmonisation:

Whenever data representations assumed by Web Services lead to semantically compatible but syntactically incongruous data flow, automated workflow harmonisation can be achieved by combining a composable, declarative mapping language with semantic service annotations, providing a scalable mediation approach that promotes sharing and reuse.

Workflow harmonisation, the act of identifying syntactic mismatches, finding the appropriate Type Adaptors, and invoking them, can be driven using data translation mediated by a canonical intermediary representation derived from the shared semantics of the service interfaces. In this dissertation, we present an architecture to support automated workflow harmonisation, making use of three principal contributions (presented graphically in Figure 1.1):

1. **Mediation**
   To enable the translation of data between different syntactic representations, a scalable mediation approach is employed making use of a declarative, composable and expressive mapping language, and a transformation implementation:

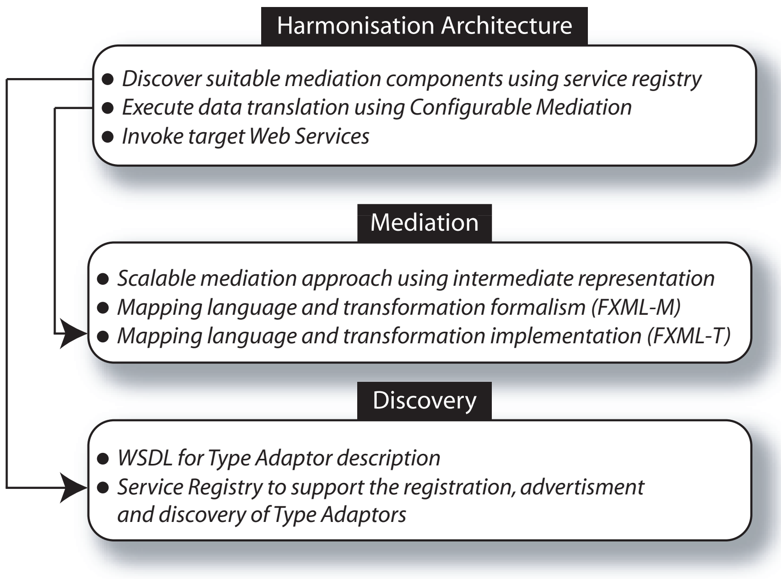

Harmonisation Architecture
Discover suitable mediation components using service registry
Execute data translation using Configurable Mediation
Invoke target Web Services
Mediation
Scalable mediation approach using intermediate representation
Mapping language and transformation formalism (FXML-M)
Mapping language and transformation implementation (FXML-T)
Discovery
WSDL for Type Adaptor description
Service Registry to support the registration, advertisment and discovery of Type Adaptors

FIGURE 1.1: A visual representation of the contributions in this Thesis.

(a) **Scalable mediation approach**

We conceived an intermediate representation, making use of OWL ontologies, to capture the structure and semantics of different data formats. With a common representation in place, maximum interoperability can be achieved by providing mappings between each data format and its corresponding OWL model. As more XML data formats are added, a linear expansion in the number of required mappings is observed.

(b) **A declarative, composable and expressive mapping language**

To specify the relationship between a concrete XML representation and its corresponding conceptual model in OWL, the bespoke mapping language FXML-M is used to define mappings that associate schema elements from a source schema to elements in destination schema using an XPATH like notation. Since complex mappings are often required, FXML-M provides predicate support (to enable the conditional mapping of elements), local scoping (so different mappings can be applied depending on element context), and the mapping of collections of elements and attributes for composite relations. Mappings are combined to form an *M*-Binding document (expressed in XML), which can be used to drive document transformation. To promote sharing and reuse,

M-Bindings may also import mappings from other documents.

(c) **A practical and scalable mapping language implementation**
FXML-T— our mapping language and transformation implementation, can be used to translate XML documents by consuming an M-Binding, the source document schema, and a destination document schema. Empirical testing proves that our Mapping Language approach is practical, our implementation scales well, M-Binding composition comes with virtually no cost, and the implementation is efficient when translating bioinformatics datasets. FXML-T is combined with the ontology reasoning API JENA [60] to create the Configurable Mediation (C-MEDIATOR): a reconfigurable Type Adaptor to enable the mediation of data through a shared conceptual model.

2. **A uniform description method for Type Adaptors using WSDL**
Because Type Adaptor components may come in many forms: e.g. translation scripts, bespoke code and Web Services, it is important to describe their capabilities uniformly. While it is understood that WSDL can be used to specify Web Service interfaces and invocation methods, we establish that Type Adaptors can also be described with WSDL, allowing existing Web Service registry technology to be reused, and support the advertising and discovery of Type Adaptor components.

3. **Automated Workflow Harmonisation Architecture**
With a configurable data translation component in place and a mechanism to specify, advertise and discover different kinds of Type Adaptors, automated workflow harmonisation can be achieved by discovering the appropriate Type Adaptors at runtime. We present our Web Services Harmonisation architecture, WS-HARMONY, that combines our mapping language implementation and Type Adaptor discovery technology to support automatic type conversion by extrapolating the conversion requirements from service definitions within a given workflow, discovering and executing the necessary Type Adaptors, and invoking the target services. Testing shows that our automated mediation approach is practical, and comes with relatively low performance cost in the context of a typical Web Service workflow execution. To invoke previously unseen Web Services, our Dynamic Web Service Invoker

(DWSI) is used, offering improvements over existing Web Service invocation APIs such as Apache Axis [10] and JAX-RPC in terms of performance and practicality.

1.2 Document Structure

We begin in Chapter 2 by investigating a bioinformatics grid application to see why workflow design and execution is impeded by service providers assuming different representations for conceptually equivalent information. Using a common bioinformatics task as a use case, we find that existing workflow harmonisation techniques are entirely manual: users must identify when mismatches in data format occur, what components are available to resolve them, and how they should be inserted into the workflow, drawing their attention away from the scientific process at hand.

In Chapter 3, we analyse related work in the areas of Semantic Web technology, data integration and automated service integration. Through assessment of the state of the art, we conclude that Semantic Web Service technology can be incorporated with existing data integration techniques to facilitate automated workflow harmonisation.

Chapter 4 presents WS-HARMONY: an architecture to support automated workflow harmonisation. The use of OWL as an intermediate representation is discussed with examples to show how our use case scenario can be harmonised using a common conceptual model. Software to support the automated discovery and execution of Type Adaptors is presented with an emphasis on the C-MEDIATOR and how it is used to create the required Type Adaptors on-the-fly.

Chapter 5 focuses on the XML data transformation problem where a formalised mapping language and transformation theory is presented in the form of FXML-M. Through the analysis of data sources within our bioinformatics use case, we derive the requirements for XML mapping and transformation which are shown to be complex.

In Chapter 6, we outline our transformation library FXML-T. This implementation of the FXML-M language is presented in detail with particular attention to the way in which rules from the formalisation are implemented. The inner workings of the C-MEDIATOR are shown, and a detailed example is provided to demonstrate how syntactic mediation is provided in our use case scenario. Empirical testing of the transformation implementation is made to establish FXML-T as scalable and efficient transformation implementation that offers M-Binding composability with virtually zero cost.

Finally, the architecture components required to make use of our C-MEDIATOR and support the automated discovery and inclusion of Type Adaptors is presented in Chapter 7. A method for the description of Type Adaptor capabilities using WSDL and their subsequent registration, advertisement, and discovery through a service registry is demonstrated along with a presentation of our Dynamic Web Service Invoker (DWSI). Conclusions and future work are given in Chapter 8 to show how our contributions can be reused in the advancement of Semantic Web Service technology.

1.3 Publications

During the development of this Thesis, the following work has been published:

Szomszor, M., Payne, T. and Moreau, L. (2005) - **Using Semantic Web Technology to Automate Data Integration in Grid and Web Service Architectures**. In *Proceedings of Semantic Infrastructure for Grid Computing Applications Workshop in Cluster Computing and Grid (CCGRID) - IEEE*, Cardiff, UK. `http://eprints.ecs.soton.ac.uk/10916/`

Szomszor, M., Payne, T. and Moreau, L. (2006) - **Dynamic Discovery of Composable Type Adapters for Practical Web Services Workflow**. In *Proceedings of UK e-Science All Hands Meeting 2006*, Nottingham, UK.
`http://eprints.ecs.soton.ac.uk/12753/`

Szomszor, M., Payne, T. and Moreau, L. (2006) - **Automated Syntactic Mediation for Web Service Integration**. In *Proceedings of IEEE International Conference on Web Services (ICWS 2006)*, Chicago, USA.
`http://eprints.ecs.soton.ac.uk/12764/`

Chapter 2

Motivation: A Bioinformatics Use Case

The Web Services computing vision promises an environment where services can be discovered, composed, executed and monitored easily. However, through the inspection of a real world Web Services application, we find that this vision has not been fully realised: the composition and execution of services is often hindered by the fact that service providers use different data formats to represent conceptually equivalent information. In order to resolve these mismatches, additional processing is required to translate data between different formats. Current solutions to this problem are entirely manual and require skilled user intervention.

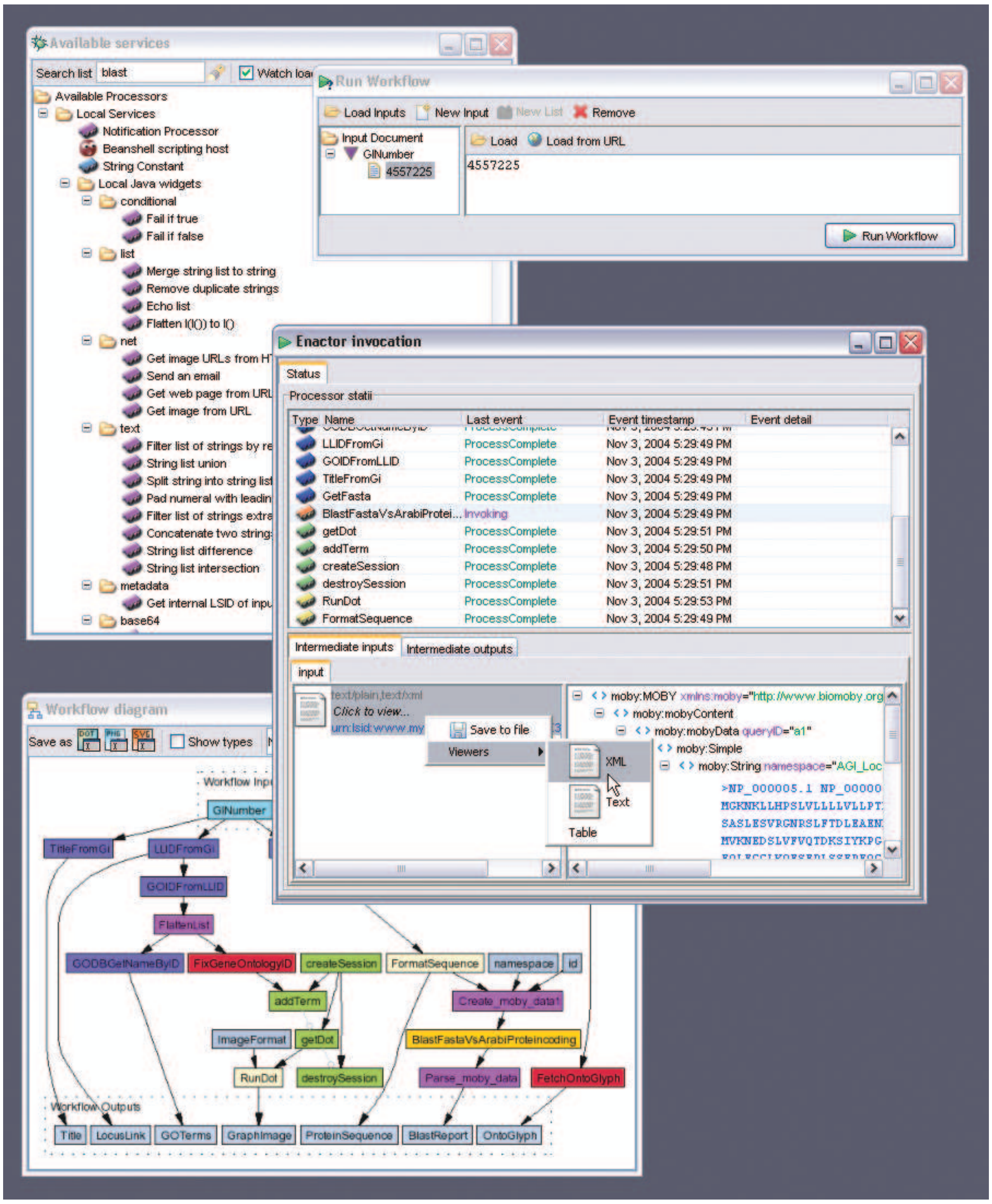
This Chapter characterises the workflow composition and execution problem, revealing the current solutions, as well as a description of a more user-friendly approach. This Chapter begins with Section 2.1, providing an introduction to bioinformatics and an overview of the myGrid [80] project. Section 2.2 follows, containing a description of how semantic annotations are used to augment the service discovery procedure. In Section 2.3, we present our use case scenario before outlining the problems it reveals in Section 2.4. Section 2.5 examines the schema reuse often employed in service interface definitions and the implications it holds for a mediation solution. We conclude in Section 2.6 by discussing the current solutions and how they can be improved.

2.1 Bioinformatics Overview

Bioinformatics is the application of computational techniques to the management, analysis, and visualisation of biological information. With the collection and storage of large quantities of genomic and proteomic data, coupled with advanced computational analysis tools, a bioinformatician is able to perform experiments and test a hypothesis without using conventional 'wet bench' equipment — a technique commonly referred to as *in silico* experimentation [49]. To support this kind of science, multiple vendors offer access to a variety of resources creating a loosely coupled, dynamic, and large scale environment which scientists can exploit to achieve their scientific aims.

The myGrid [80] project provides an open-source Grid middleware that supports *in silico* biology. Using a service-oriented architecture, a complex infrastructure has been created to provide bioinformaticians with a virtual workbench with which they can perform biological experiments. Access to data and computational resources is provided through Web Services which can be composed using the workflow language XSCUFL [97] and executed with the FreeFluo [46] enactment engine. The biologist is provided with a user interface (Taverna [89]) which presents the services available, enables the biologist to compose and view workflows graphically, execute them, and browse the results. A screenshot of the Taverna workbench is shown in Figure 2.1 and contains four windows: Available Services, Workflow Diagram, Run Workflow, and Enactor Invocation.

The **Available Services** window in the top left shows a list of services the user has access to and the operations each service offers. The **Workflow Diagram** window in the bottom left shows a graphical representation of the current workflow. Each box represents a service invocation and the arrows joining them represent the flow of data. The user is able to drag and drop services from the available services list into the graphical editor to add a service to the current workflow. The graphical representation of the workflow is mirrored in XML in the form of an XSCUFL workflow document. The **Run Workflow** and **Enactor Invocation** windows enable the user to view the workflow's invocation steps and any intermediate results, as well as the status of any currently running processes.



Type	Name	Last event	Event timestamp	Event detail
	LLIDFromGi	ProcessComplete	Nov 3, 2004 5:29:49 PM	
	GOIDFromLLID	ProcessComplete	Nov 3, 2004 5:29:49 PM	
	TitleFromGi	ProcessComplete	Nov 3, 2004 5:29:49 PM	
	GetFasta	ProcessComplete	Nov 3, 2004 5:29:49 PM	
	BlastFastaVsArabiProtei...	Invoking	Nov 3, 2004 5:29:49 PM	
	getDot	ProcessComplete	Nov 3, 2004 5:29:51 PM	
	addTerm	ProcessComplete	Nov 3, 2004 5:29:50 PM	
	createSession	ProcessComplete	Nov 3, 2004 5:29:48 PM	
	destroySession	ProcessComplete	Nov 3, 2004 5:29:51 PM	
	RunDot	ProcessComplete	Nov 3, 2004 5:29:53 PM	
	FormatSequence	ProcessComplete	Nov 3, 2004 5:29:49 PM	

FIGURE 2.1: The Taverna Workbench.

Within the Taverna application, one of the most difficult tasks the user faces is finding the service instances they require. Typically, the user has planned their experimentation process prior to their interaction with the Taverna workbench. Most likely, this has been done on paper with various abstract definitions of the types of service required for each stage in the process. To find a particular service which achieves a goal they desire, the user has to inspect the services available in the **Available Services** window and manually choose the most appropriate one. Given the terse and often cryptic service descriptions, and the sheer number of services offered (over 1000 in myGrid) [58, 71], the discovery of services is awkward. Hence, recent research from the myGrid project has been centred on the incorporation of *Semantic* discovery.

2.2 Semantic Discovery

According to the basic premise of the Semantic Web [20], information should be presented in a way where the meaning is captured in a machine processable format so computers can understand the semantics of the data and exchange information accurately. This vision has been partially realised through the use of ontologies: a language to formally define a common vocabulary of terms for a given domain and how such terms relate to each other [51], and in particular, through the development of ontology languages, such as OWL [83], that provide mechanisms to support the publishing, sharing, and collaboration of ontology definitions using the Web. By annotating service definitions with concepts from these shared ontologies, users can find services based on conceptual definitions of the service capability, rather than the low-level interface definitions.

Figure 2.2 is a graphical representation showing part of an ontology created to capture the kinds of terms used in the description of bioinformatics data [94]. With this domain model in place, and the appropriate service annotations, a bioinformatician can discover services according to the task it performs (e.g. retrieving, processing or visualising data), the resources it uses (e.g. particular databases), and the type of inputs and outputs (e.g. consumes sequence data and produces an alignment result), rather than simply the labels used or the data types specified in

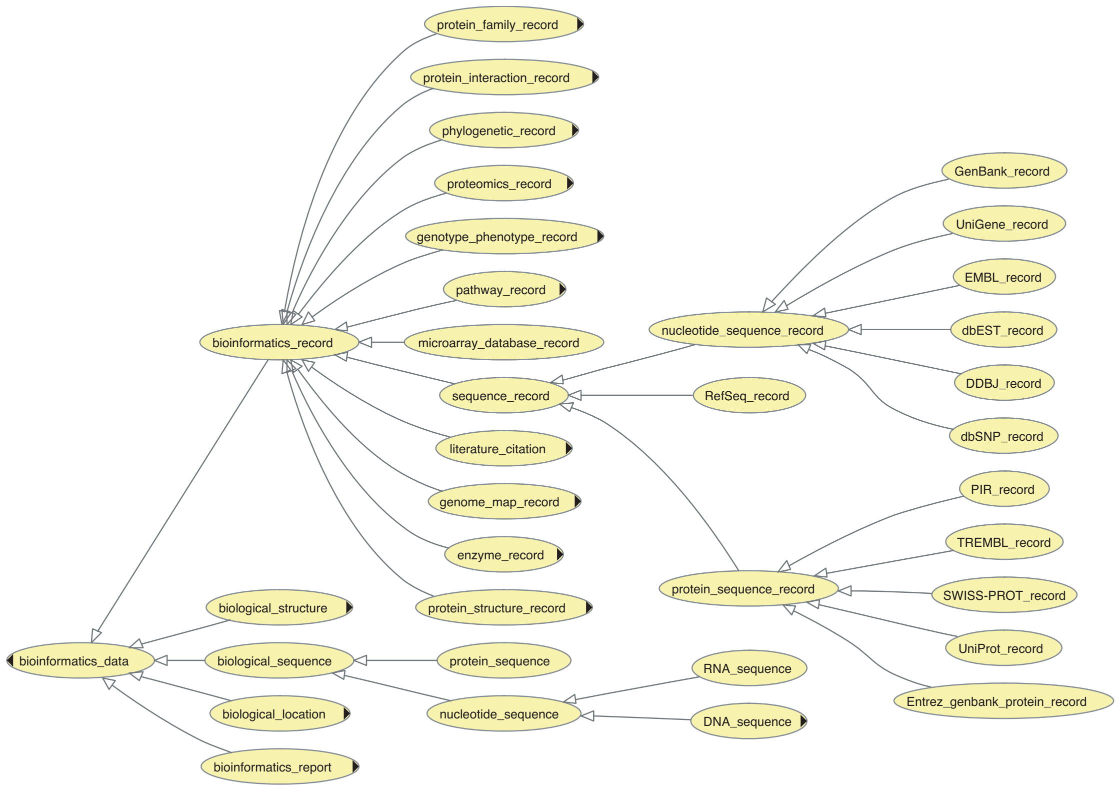
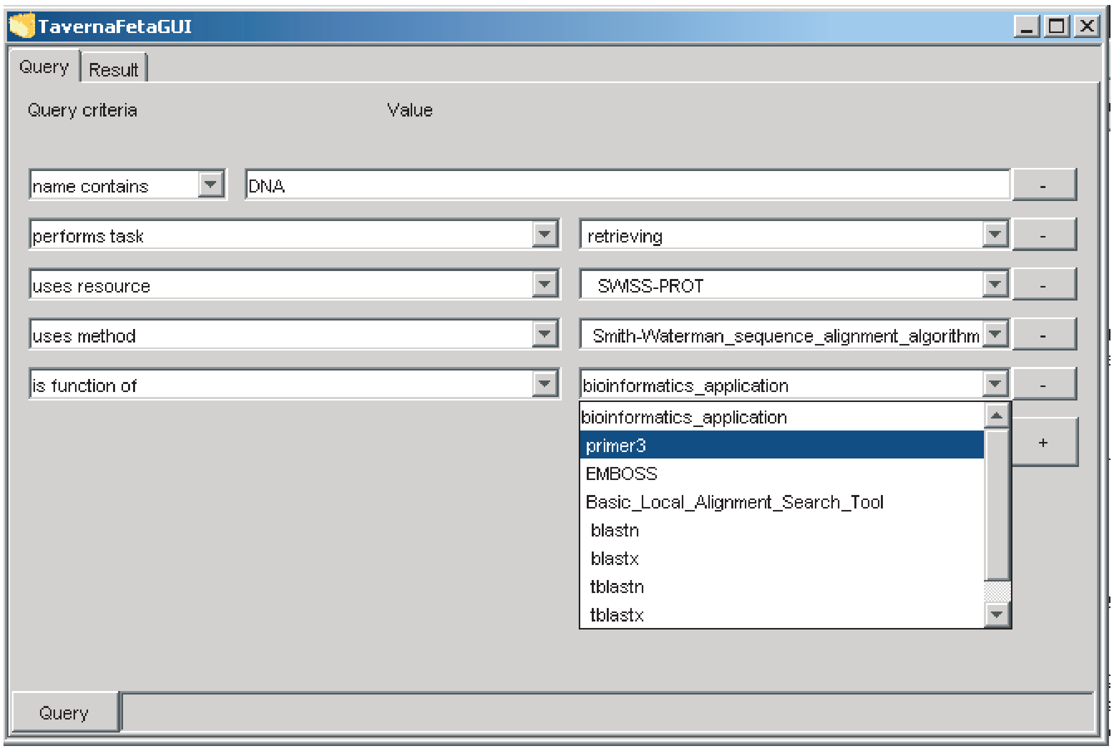


FIGURE 2.2: A subset of the bioinformatics ontology developed by the MYGRID project.

the service interface definition. To implement this feature in Taverna, the Pedro [48] tool is used to annotate service definitions with concepts from the bioinformatics ontology. These semantic annotations can then be consumed by FETA [71], a light-weight architecture for user-oriented semantic service discovery, that in turn, provides a query interface to search over services. The most recent release of the Taverna workbench provides a graphical query interface to FETA, which is illustrated in Figure 2.3. The query window here allows the user to find services based on values of certain service attributes which have been set previously using the Pedro annotation tool. In this example, some of the required attributes are:

1. the service name must contain the string `"DNA"`;
2. it must perform the task with concept `"retrieving"`;
3. it must make use of the `"SWISS-PROT"` resource (a database in this case).

Any service instances matching those criteria will be returned to the user when the query is submitted.



Query criteria	Value
name contains	DNA
performs task	retrieving
uses resource	SWISS-PROT
uses method	Smith-Waterman_sequence_alignment_algorithm
is function of	bioinformatics_application

bioinformatics_application
primer3
EMBOSS
Basic_Local_Alignment_Search_Tool
blastn
blastx
tblastn
tblastx

FIGURE 2.3: FETA Discovery Tool within Taverna

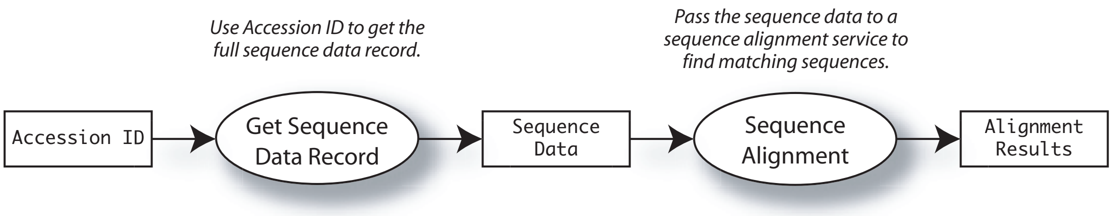

Use Accession ID to get the full sequence data record.
Pass the sequence data to a sequence alignment service to find matching sequences.
Accession ID
Get Sequence Data Record
Sequence Data
Sequence Alignment
Alignment Results

FIGURE 2.4: An abstraction view of our bioinformatics use case

2.3 Use case

For our use case, we examine a typical bioinformatics task: *retrieve sequence data from a database and pass it to an alignment tool, such as Blast [6], to check for similarities with other known sequences.* According to the service-oriented view of resource access adhered to by MYGRID, this interaction can be modelled as a simple workflow with two stages: an initial stage to retrieve the sequence data, and a second stage to check for similarities with other sequences. We show this simple workflow at an abstract level in Figure 2.4.

To turn this abstract definition into a concrete workflow, the user must discover suitable services for each step. Many Web Services are available to retrieve sequence data, for our example, we could use one available from EMBL [96], or alternatively, one from DDBJ [36]. For the second stage of the workflow, an alignment service available from NCBI [81] could be used. Therefore, two concrete workflows can be created to perform the analysis: one using the XEMBL service and another using the DDBJ-XML service, illustrated in Figure 2.5.

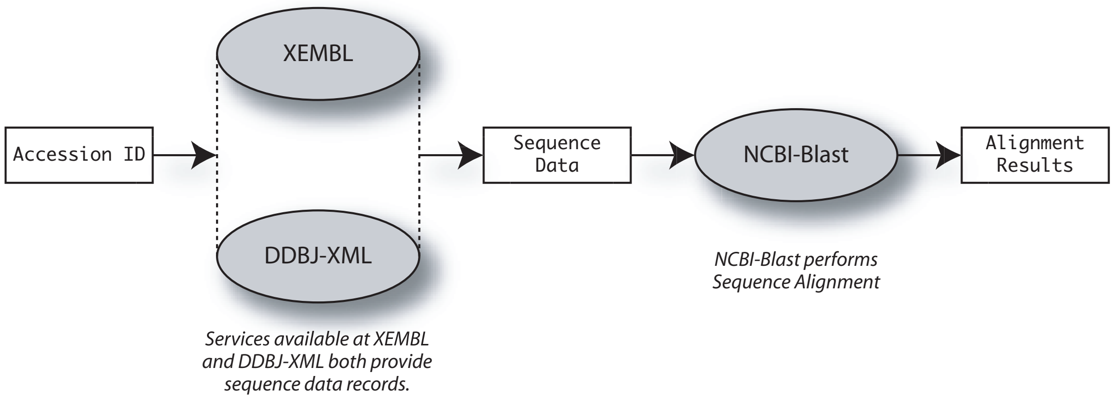
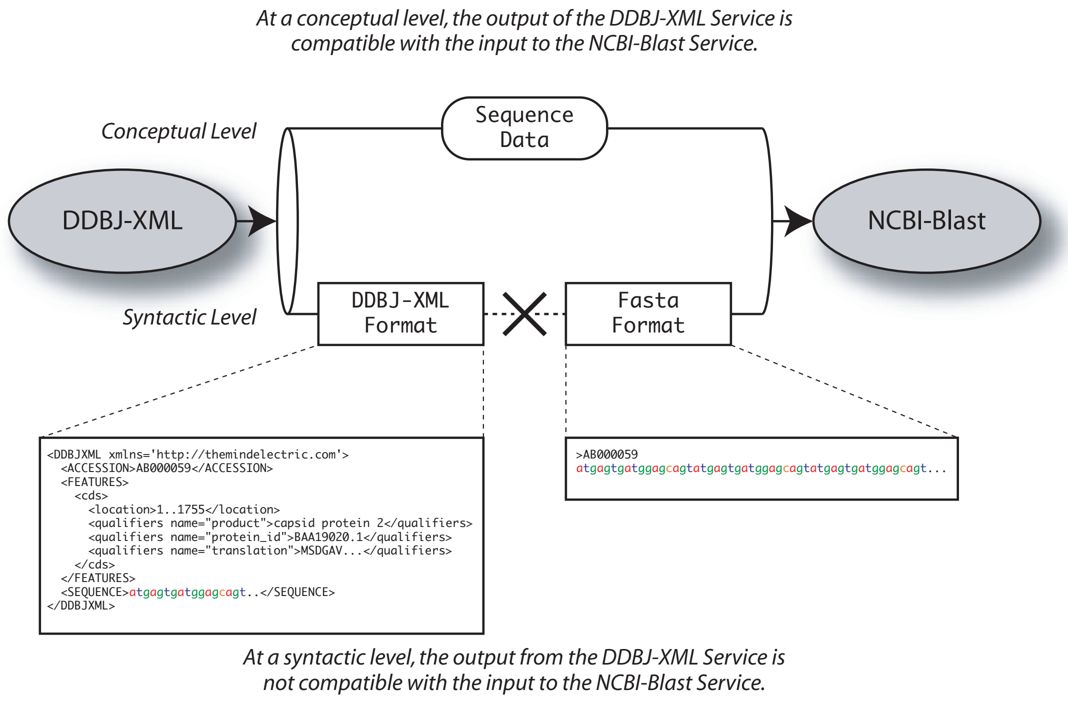

XEMBL
Accession ID
Sequence Data
NCBI-Blast
Alignment Results
DDBJ-XML
NCBI-Blast performs Sequence Alignment
Services available at XEMBL and DDBJ-XML both provide sequence data records.

Figure 2.5: Two possible concrete workflows for a sequence retrieval and analysis task.

2.4 Syntactic Compatibility

While both sequence retrieval services are similar, in that an accession id is passed as input to the service and an XML document is returned containing all the sequence data, the format of the XML documents is different: XEMBL returns an EMBL-EBI formatted document[1], whereas DDBJ-XML returns a document using their own custom format[2]. When considering the compatibility of the data flow between the services, it can be seen that the output from neither sequence retrieval service is directly compatible for input to the NCBI-Blast service. Figure 2.6 illustrates this example: the DDBJ-XML service produces a DDBJ formatted sequence data record, and the NCBI-Blast service consumes a FASTA formatted sequence.

[1]http://www.ebi.ac.uk/embl/schema/EMBL_Services_V1.0.xsd
[2]http://getentry.ddbj.nig.ac.jp/xml/DDBJXML.dtd


At a conceptual level, the output of the DDBJ-XML Service is compatible with the input to the NCBI-Blast Service.
Conceptual Level
Sequence Data
DDBJ-XML
NCBI-Blast
Syntactic Level
DDBJ-XML Format
Fasta Format
At a syntactic level, the output from the DDBJ-XML Service is not compatible with the input to the NCBI-Blast Service.

```
<DDBJXML xmlns='http://themindelectric.com'>
  <ACCESSION>AB000059</ACCESSION>
  <FEATURES>
    <cds>
      <location>1..1755</location>
      <qualifiers name="product">capsid protein 2</qualifiers>
      <qualifiers name="protein_id">BAA19020.1</qualifiers>
      <qualifiers name="translation">MSDGAV...</qualifiers>
    </cds>
  </FEATURES>
  <SEQUENCE>atgagtgatggagcagt..</SEQUENCE>
</DDBJXML>
```

```
>AB000059
atgagtgatggagcagtatgagtgatggagcagtatgagtgatggagcagt...
```

FIGURE 2.6: The DDBJ-XML output is conceptually compatible with the input to the NCBI-Blast service, but not syntactically compatible.

To execute a workflow where the output from the DDBJ-XML service is passed as input to the NCBI-Blast service, the differences in data format assumed by each provider must be reconciled, a process we refer to as *workflow harmonisation*. Within Taverna, this is a manual task: users must identify when a syntactic mismatch has occurred, what components are available to carry out the necessary translation, and in many cases, new ones must be created. The transformation of data between different representations by an external software components, or *syntactic mediation*, can be achieved using a variety of techniques: a bespoke mediator could be programmed using a language such as JAVA; a transformation language such as XSLT [34] is used to specify how the translation is performed; or another mediation Web Service could be invoked. We use the term *Type Adaptor* to describe any software component that enables the translation of data, either declaratively (in the case of a script) or procedurally (in the case of a program or Web Service).

2.5 Data format reuse in Web Services

Upon further examination of services in myGrid, it is apparent that a single Web Service may offer a number of different operations. The service interface definition (expressed using WSDL [33]) defines the input and output types for each operation by referencing an XML schema type (referred to as the *syntactic type*), and semantic annotations attached using the Pedro tool define the conceptual type for each input and output by referencing a concept from the bioinformatics ontology (the *semantic type*). Often, it is the case that a Web Service offers operations with inputs and outputs that utilise parts of the same global data structure. For example, the DDBJ-XML Service in our use case offers many operations over sequence data records. A single XML schema exists to describe the format of the sequence data record, and each operation defines the output type by referencing an element within this schema. We illustrate this scenario in Figure 2.7 by showing the DDBJ-XML interface definition (right box), and how semantic annotations relate message parts to concepts from the bioinformatics ontology (left box).

The DDBJ-XML service offers access to many sequence data repositories: SWISS, EMBL and UNIPROT are three of them. Each of these databases is maintained separately so users may elect to retrieve sequence data from one source over another. To support this, the DDBJ-XML service offers separate operations to supply

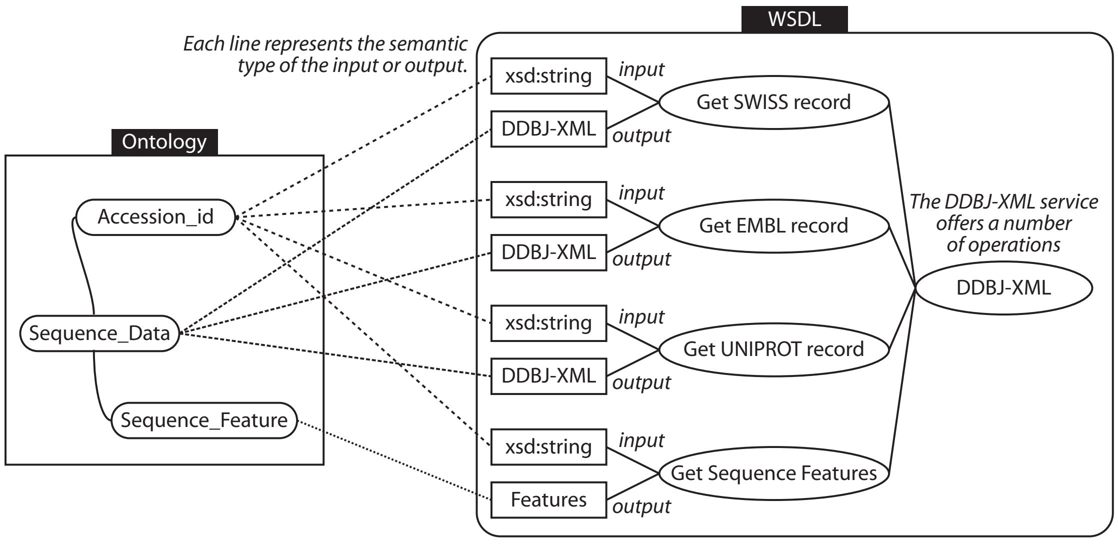


Figure 2.7: The DDBJ-XML web service offers a number of operations over the same XML schema.

access to each repository. Each of these three operations has the same input and output types, both in terms of the syntactic type and the semantic type. The DDBJ-XML service also offers operations to retrieve only parts of the sequence data record. In Figure 2.7, we show the "Get Sequence Features" operation that allows users to retrieve only the features of a sequence data record. In this case, the output syntactic type is the `<Feature>` element from the DDBJ-XML schema and the semantic type is the *Sequence_Feature* concept. With this kind of schema reuse, one can imagine that a single Type Adaptor component may be suitable for use with many service operations, even those which operate over a subset of a global schema. In our use case, a single Type Adaptor could translate sequence data from DDBJ-XML format to FASTA format, and would therefore be suitable for any of the operations shown in Figure 2.7.

2.6 Conclusions

To achieve workflow harmonisation in a workflow-driven service-oriented environment that encourages users to discover services through high-level conceptual descriptions, some form of syntactic mediation is often required to translate data between different representations. Current solutions require users to find manually manually (or possibly create) any required translation components. Given the wide variety of heterogeneous services typically offered in large-scale eScience applications, syntactic mediation components often take up a significant proportion of the composition when compared to the actual services required to achieve experiment goals. Naturally, this hinders the scientific process because users frequently spend time harmonising the services in their composition rather than actually designing and using it. Furthermore, it contradicts the basic Semantic Web Service ideology because users must consider low-level interoperability issues. Therefore, our aim is to automate the process of workflow harmonisation so users can create meaningful workflows without concern for the interoperability issues that arise from heterogeneous data representations. This means the identification of syntactic mismatches, discovery of Type Adaptors, and their execution must be addressed. As we highlighted in Section 2.5, services often provide operations that consume or produce information using the same or subsets of the same data formats. Consequently, an automated workflow harmonisation approach should embrace the reuse of format specification, which in turn, can reduce the chances of Type Adaptor duplication.

Chapter 3

Background

In Chapter 2, we analysed a bioinformatics Grid application and found that workflow composition and execution is often hindered by the differences in data representation assumed by service providers. Our aim is to improve on the current manual solutions and support autonomous workflow harmonisation through the discovery and invocation of necessary Type Adaptors at runtime. This Chapter examines background material in the areas of Grid, Web Services, and Semantic Web, as well as a review of related work in the fields of data integration, service integration and workflow composition.

We begin in Section 3.1 with an introduction to Grid and Web Services, summarising the fundamental technologies and their limitations. Section 3.2 shows how Semantic Web technology can augment existing Grid and Web Service environments, supporting more intuitive service discovery and facilitating autonomous service invocation. We describe the current technologies that aim to support the application of Semantic Web techniques to Web Service architectures and finish with a comparison of their approaches.

In Section 3.4, an overview of relevant data integration work is presented, highlighting the similarity of problems addressed with those underpinning the workflow harmonisation problem. We present existing workflow harmonisation technology in Section 3.6 and discuss the relevance to our workflow harmonisation problem. In Section 3.7, we look into high-level service discovery and workflow composition techniques to see how they support the creation of meaningful workflows in scientific applications. We conclude in Section 3.8 by discussing how data integration approaches can be combined with Semantic Web Service technology to support automated service integration.

3.1 Grid Computing and Web Service

Ever since the early 1970's, when computers were first linked together by networks, the concept of harnessing computational power from multiple machines to perform a single task has become a fundamental computer science field. Research in this field has been driven mainly by the high performance computing community who concentrate on splitting up large computational tasks, allocating them to multiple machines for calculation, and reintegrating the final results.

In the 1990's the distributed computing community saw new opportunities arise through the emergence of the Internet. The Internet provides a global network on which any two machines on the planet can communicate across a simple and reliable transport mechanism. This explosion in connectivity and computing power has been matched by the increasing complexity of tasks users want to perform. Particularly motivated by scientific fields such as astronomy, particle physics and biology, users now demand access to powerful computational systems that hold vast amounts of data collected from a range of disparate sources.

The Grid is a distributed computing infrastructure designed to support exactly this type of complex behaviour: co-ordinated resource sharing across dynamic and geographically separated organisations [44]. This resource sharing covers a wide range of computing assets including computational power, information storage and observational equipment. With this variety of heterogeneous resource types exposing a diverse mix of functionality, a fundamental problem that must be addressed by the Grid is how to provide a homogeneous access model to *all* types of resource. For example, whether a resource exposes data stored within a database or software which processes data, a homogeneous resource description and access model must be used to ensure maximum interoperability. This issue was addressed by the Open Grid Services Architecture (OGSA) [45] where a service-oriented view of resource access is employed. Essentially, this means that access to every type of resource is modelled as though it is a service. To implement this type of architecture, Web Services can be used.

3.1.1 Web Service Architecture

The Grid architecture relies on a service-oriented view of resource access inspired by the use of the Web Services Architecture [23]. This allows resource providers to describe their capabilities in terms of a Web Service, most commonly through the use of WSDL [33]. WSDL is a specification language that describes the abstract operational characteristics of a service using a message-based paradigm. Services are defined by operations (which are analogous to methods from traditional programming paradigms), each of which has an input message and an output message (like the parameters and result of a method). Each message may contain a number of parts, each of which is defined by a reference to a schema type (typically XML Schema [41]). These abstract definitions are bound to concrete execution models to explain the invocation method (for example SOAP [52] encoding over HTTP transport).

Other Web Service technologies are also available to support more intricate service functionality such as service discovery (UDDI [1]), secure message exchange (WS-Security [61]) and the specification of collaborations between resources (WS-CDL [62]). Many software implementations are available to support the Web Services Architecture, such as the Apache Axis Web Service Project [10] and IBM's WebSphere Software suite [59]. To this end, the Web Service Architecture provides a fundamental model on which to build Grid computing applications through a set of widely recognised standards and a range of software tools to support them.

3.1.2 Web Service Limitations

While the use of WSDL provides us with a common way to view the invocation parameters of a Web Service, such as the format of a valid message and the concrete execution model, it does not supply any information on what the service does - a notion usually referred to as the *semantics* of the service. This leads to two major problems:

1. **Unsophisticated service discovery**
   Service discovery is the process through which we can find services that perform a given task. Although current Web Service standards such as UDDI and ebXML [40] support the registration and indexing of large numbers of services, their information models are constrained, allowing only simple string matching on business and service types. In extreme cases, interface definitions are completely undocumented and operation names bear little relation to the actual functionality provided. Paolucci *et al* [82] demonstrates that human comprehension of a service description is required before we can be sure that it matches any given requirements. Hence, the level of autonomy we can achieve is limited.

2. **Limited automatic invocation**
   Assuming a candidate service has been discovered, we would then wish to invoke it. WSDL describes the structure and required data types of the message contents, so we can determine a valid message format. However, it does not state what the parts of the message contents represent. For example, a service may expose an operation to calculate the distance between two cities. The interface for such a service could take two strings as input: one for the source city name and one for the destination city name. Without additional semantics, an automated invocation component would not know which city name to place in which part since they both have the same type, namely a string. Since service vendors are unlikely to subscribe to a predefined message layout, we cannot assume that a client will know how to create the correct message [77].

To overcome these problems we require additional high-level service descriptions (service annotations) that express the service behaviour in an unambiguous, machine interpretable format. Expressing service properties in such a manner is commonly referred to as a *semantic* service description or *formal semantic model.*

3.2 Web Services and Semantics

The *Semantic Web* [20] is an extension of the existing Web that aims to support the description of Web resources in formats that are machine understandable. According to the Semantic Web approach, resources are given well-defined meaning by annotating them with concepts and terminology that typically correlate with those used by humans. To share knowledge at a high-level using well-defined semantics, we can use an ontology: a modelling language to formally define a common vocabulary of terms for a given domain and how such terms relate to each other [51]. The Web Ontology Language (OWL) [83] is an example of an ontology language that is specifically designed to facilitate the publication and sharing of domain models on the Web. OWL is an extension of the existing Resource Description Framework (RDF) [66], incorporating features from the DARPA Agent Markup Language (DAML) [91] and Ontology Inference Layer (OIL) [56] to create a rich conceptual modelling language with well defined reasoning procedures founded on description logics [12].

In OWL, *classes* (or concepts) are used to define groups of items or individuals that should be collected together because they share common attributes. For example, the individuals Anne, Barry, and Colin would all be members of the Person class. *Properties* are used to state relationships between classes of individuals, (called *object properties*, and from classes to specific data values (called *datatype properties*). Class hierarchies can be used to specify classes which are considered more specific or general than other classes (e.g. the class Male is more specific than the class Person). Individuals (or *concept instances*) can then be specified in terms of defined classes with object properties to relate them to other individuals, and datatype properties that define their attributes using literal values such as strings or integers. With OWL, both the ontology definition and its concept instances can be represented using an XML syntax, with datatype properties instantiated using XML schema types. Through the use of ontologies, the Semantic Web supplies computers with rich annotations enabling them to reason on resources distributed across the Web through a common, high-level conceptual model.

By applying the Semantic Web approach to a Web Services architecture, existing Web Service interface definitions can be annotated with semantic descriptions.

This approach supports: (a) more advanced service discovery [72] because queries on a service's functionality can be formulated in terms of the high level, human oriented descriptions; and (b) better automation [9] because service interfaces will be annotated with semantics that describe what the data represents and not just its syntactic type.

To enable the annotation of Web Services with semantics, the use of ontologies is critical. Given that any particular service instance operates within a set of domains (e.g. a book buying service works in the purchasing and book information domains), we can encode the operational characteristics of the service using an ontology. For example, a purchasing ontology would have concepts describing payment, shipping, ordering, etc. and a book information ontology would describe books, authors, publishers, etc. To create these service description ontologies, we must ensure that we encapsulate the necessary information:

- **Information processing**
  What are the inputs and outputs of the service? Given that service providers will often use their own bespoke data structures, an ontology describing the service information requirements should state what the inputs and outputs are using terms from a shared conceptualisation, enabling clients to determine what each part of a service's interface means. This enables clients to invoke services properly by ensuring that data given to the service for processing is both the correct syntactic type (specified in the interface definition) and appropriate semantic type (the concept referenced within the ontology).

- **Behaviour**
  How does this service change the state of the system? Many services have effects other than the immediate processing of data. For example, a service which allows a customer to purchase an item needs to represent the notion that after a successful invocation the customer's credit card account will be reduced by a certain amount and the requested item will be shipped to them. Capturing this behaviour is essential since two different services that consume and produce conceptually equivalent data need to be distinguished from each other by the effects they have on the real world.

The combination of Semantic Web technology with Web Services to produce *Semantic Web Services* has received a great deal of attention. In the following subsections, we investigate the major technologies that aim to support the Semantic Web Service vision, before comparing their approaches.

3.2.1 OWL-S

OWL-S is a set of ontology definitions (expressed using the OWL ontology language) designed to capture the behaviour of Web Services. The top level *service* ontology presents the service *profile*, a description of what the service does (e.g. that a service is used to buy a book). The service is described by the service *model*, which tells us how the service works (e.g. a book buying service requires the customer to select the book, provide credit card details and shipping information and produces a transaction receipt). Finally, the service supports the service *grounding* that specifies the invocation method for the service. Figure 3.1 shows the basic relationship between these top level ontologies. In terms of data representation and service invocation, our interest lies primarily in the service grounding ontology because it describes the relationship between the high-level service description, encapsulated within the OWL-S ontology definition of the service, and the actual service interface.

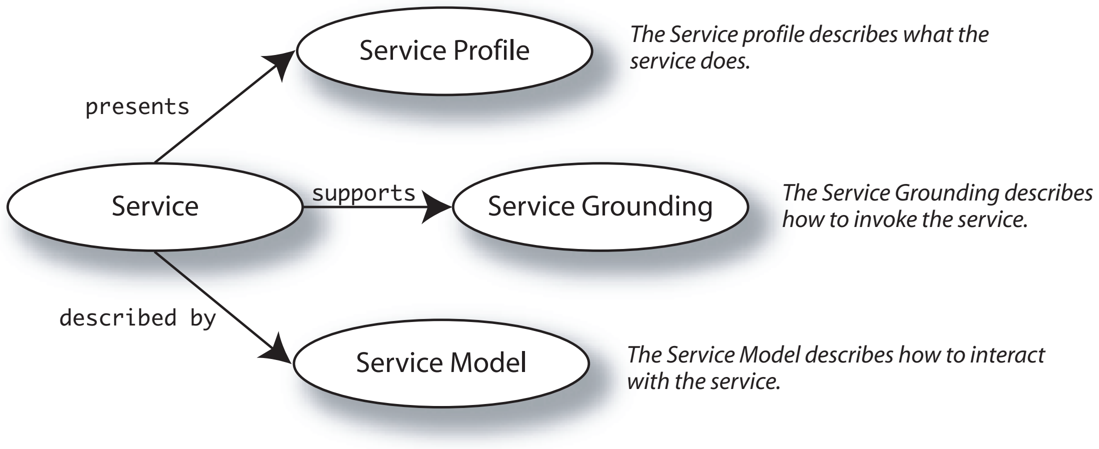

Service Profile
The Service profile describes what the service does.
presents
Service
supports
Service Grounding
The Service Grounding describes how to invoke the service.
described by
Service Model
The Service Model describes how to interact with the service.

FIGURE 3.1: OWL-S services are described by three facets; a profile, a grounding and a model

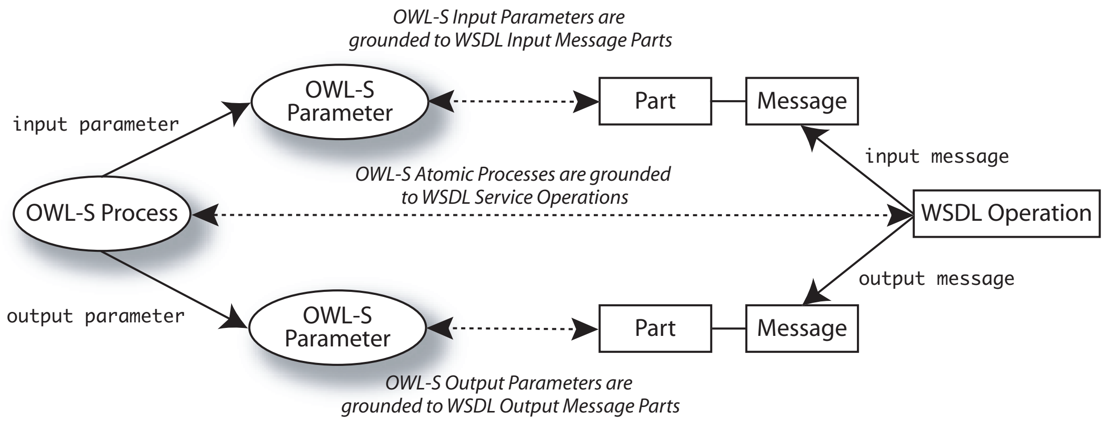

OWL-S Input Parameters are grounded to WSDL Input Message Parts
OWL-S Parameter
Part
Message
input parameter
input message
OWL-S Atomic Processes are grounded to WSDL Service Operations
OWL-S Process
WSDL Operation
output message
output parameter
OWL-S Parameter
Part
Message
OWL-S Output Parameters are grounded to WSDL Output Message Parts

FIGURE 3.2: OWL-S atomic processes are grounded to WSDL Service operations. Each OWL-S parameter is grounded to a WSDL message part.

The current OWL-S release (Version 1.2 Beta [2]) supports the annotation of WSDL interfaces for services that use SOAP invocation only. The basic grounding premise is that each OWL-S atomic process (the most basic process element) is grounded to a WSDL operation. The input parameters to an atomic process, which represent the conceptual type of the input, are grounded to WSDL input message parts. The same applies for the output parameters: they are grounded to WSDL output message parts (Figure 3.2).

To annotate an existing Web Service that has a WSDL definition, XSLT scripts are used to describe how an input OWL concept instance (serialised in XML) is translated to a SOAP envelope so it can be sent directly to the service for invocation. The reverse applies for the service output: XSLT is used to translate the output SOAP envelope into an OWL concept instance. While this is a rather restrictive approach since only one style of Web Service invocation and data encoding is supported, it is only one implementation style that has been explored by the OWL-S community: OWL-S is designed to be extensible and support other encoding types and invocation styles, although this has yet to be explored.

3.2.2 WSMO

The Web Services Modelling Ontology (WSMO) [84] is an evolving technology built upon, and extending the earlier UPML [43] framework. WSMO is designed to provide a framework to support automated discovery, composition, and execution of

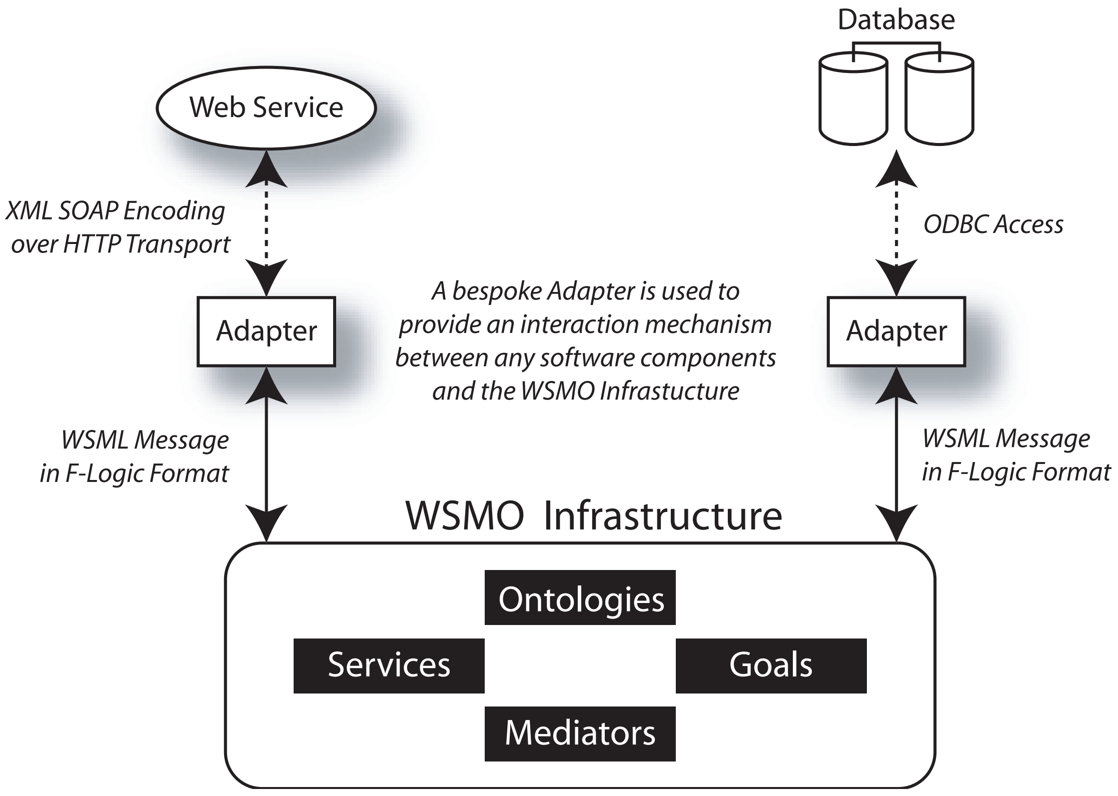

Database
Web Service
XML SOAP Encoding over HTTP Transport
ODBC Access
Adapter
A bespoke Adapter is used to provide an interaction mechanism between any software components and the WSMO Infrastucture
Adapter
WSML Message in F-Logic Format
WSML Message in F-Logic Format
WSMO Infrastructure
Ontologies
Services
Goals
Mediators

Figure 3.3: With wsmo, adaptors are placed in front of legacy components, such as Web Services, to provide a bridge to the wsmo infrastructure.

Web Services based on logical inference mechanisms. Conceptually, wsmo is based on an event driven architecture so services do not directly invoke each other, instead goals are created by clients and submitted to the wsmo infrastructure which automatically manages the discovery and execution of services. Like owl-s, wsmo uses ontologies to describe both the behaviour of Web Services and the information they consume and produce. This is achieved using the bespoke F-Logic based language wsml [37]. It is assumed that components within the wsmo architecture communicate using a standardised message format: an xml serialisation of the wsml language. Essentially, this means that all participants within a wsmo framework are expected to communicate at a conceptual level using xml serialisations of wsml concepts. To accommodate differences in conceptual representation, the wsmo infrastructure also contains explicit mediator components that support the translation of information between different wsml representations.

To elevate conventional computing resources, such as Web Services and databases, into the wsmo framework, message adaptors are placed in-front of the resource to deal with the translations to and from traditional syntactic interfaces (such as a SOAP interface to a Web Service or an ODBC interface to a database) and the WSML message layer as we show in Figure 3.3 These *Adaptors* are a super set of what we defined earlier as Type Adaptors because they are responsible for more than the translation of data between different syntactic representations: conversions between different access models (e.g. relational databases and XML data), different transport types (e.g. HTTP, and FTP), and different interaction protocols (e.g. request / response Web Services, and remote method invocation). An example of such an adaptor can be found in Section 5.3 of [76] which performs translations between WSML and Universal Business Language [74] (UBL). With this approach, the syntactic interface to a business service is hidden because its interface is exposed only through the WSMO framework.

3.2.3 WSDL-S

WSDL-S (Web Service Semantics, a W3C Member submission) [5] is an extension of the existing WSDL interface definition language that supports meta-data attachment. WSDL-S assumes formal semantic models (i.e. models that describe the service behaviour using semantics) exist outside the WSDL document and are referenced via WSDL extensibility elements. WSDL-S is technology agnostic so any formal semantic model can be used, such as OWL-S or WSMO. We provide a visual representation in Figure 3.4 that shows a conventional WSDL document referencing an OWL ontology (to provide formal semantics for the data types) and an OWL-S definition (formal semantics for the service behaviour). To support the relationship between concrete data (in XML) and its conceptual representation (in OWL), WSDL-S has two annotation models:

1. **Bottom Level Annotation**
   For simple cases, when a one-to-one correspondence exists between an XML element within the WSDL message and an OWL concept, bottom level annotations can be used to specify the mapping by means of an extensibility element. While this model is limited (complex types are not supported), it is sufficient for many cases.

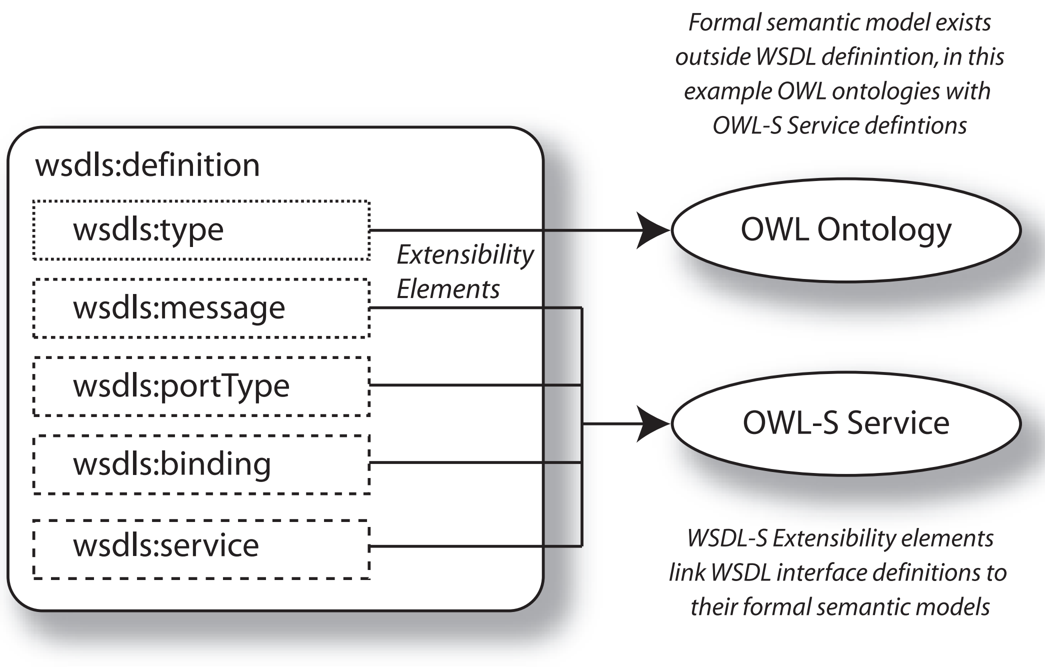

Formal semantic model exists outside WSDL definintion, in this example OWL ontologies with OWL-S Service defintions
wsdls:definition
wsdls:type
wsdls:message
wsdls:portType
wsdls:binding
wsdls:service
Extensibility Elements
OWL Ontology
OWL-S Service
WSDL-S Extensibility elements link WSDL interface definitions to their formal semantic models

FIGURE 3.4: WSDL-S annotation approach: WSDL Definitions are linked to external semantic models via extensibility elements

2. **Top Level Annotation**
   With the top level annotation approach, an external mapping can be referenced that specifies the full translation between XML and OWL. This allows complex data representations to be assigned a model in OWL. Again, WSDL-S is technology agnostic so any form of mapping can be used, such as XSLT or XQUERY [22].

3.2.4 Comparison of Semantic Annotation Techniques

OWL-S, WSMO, WSDL-S and the annotation policy adopted by FETA (discussed previously in Chapter 2) are oriented around the idea of high-level service descriptions specified using an ontology based language: OWL in the case of OWL-S, WSML for WSMO, and RDFS [25] for FETA. The description approaches are similar: inputs and outputs to services are specified using concepts from an ontology describing the domain in which the service operates. Changes to the state of the world are defined using pre-condition and effect based constructs, i.e. some state of the system must be true before execution is permitted and successful invocation results in new facts being added. The difference in approach lies fundamentally in their implementation methodology. OWL-S and FETA are used as an annotation model, supplying language constructs to describe the behaviour of services at a conceptual level without imposing any standard message exchange format or invocation style. While current OWL-S implementations are based around somewhat restricted models, i.e. WSDL interface annotation with SOAP invocation, the model is designed to be extensible and therefore support other types of Web resource and access methods. Implementation of the WSMO architecture can be considered more mature than those supporting OWL-S. The WSMX framework which implements WSMO already supplies a integrated annotation, discovery and invocation environment - something which has yet to be fully realised by the OWL-S community. However, this has been achieved mainly because of the restrictions placed on WSMO participants, namely a standardised message exchange format and imposed invocation style. Since WSDL-S is only an annotation approach that relies on the existence of an external formal model, it cannot be compared directly to OWL-S, WSMO, or FETA. However, it does subscribe to the same basic principle i.e. services are described using high-level, conceptual definitions expressed in an ontology.

3.3 Viewing a Semantic Web Service Architecture

The amalgamation of the term "semantic" with "web service" to produce *Semantic Web Service* has been used frequently, but also indiscriminately. Sometimes it is used to refer to the notion that existing Web Services are augmented with Semantic Web technology to aid computers in understanding what the service does and how it works. All other times, it used to refer to a new type of service that sits on the Semantic Web, directly consuming and producing information at the conceptual level. To distinguish between these different views, we introduce the terms *semantically annotated* Web Service and *semantically enabled* Web Service.

Semantically annotated Web Services are conventional Web Services, such as those described by WSDL, that have been annotated with a semantic description. This allows the service to continue interacting with traditional clients, as well as allowing more advanced components, such as a discovery service, to utilise the additional annotations and reason on the capabilities of the service. We illustrate this type of service in Figure 3.5. With these types of service, some mechanism must exist to describe how conceptual information structures, such as ontology instances, are grounded to concrete data representations such as XML.

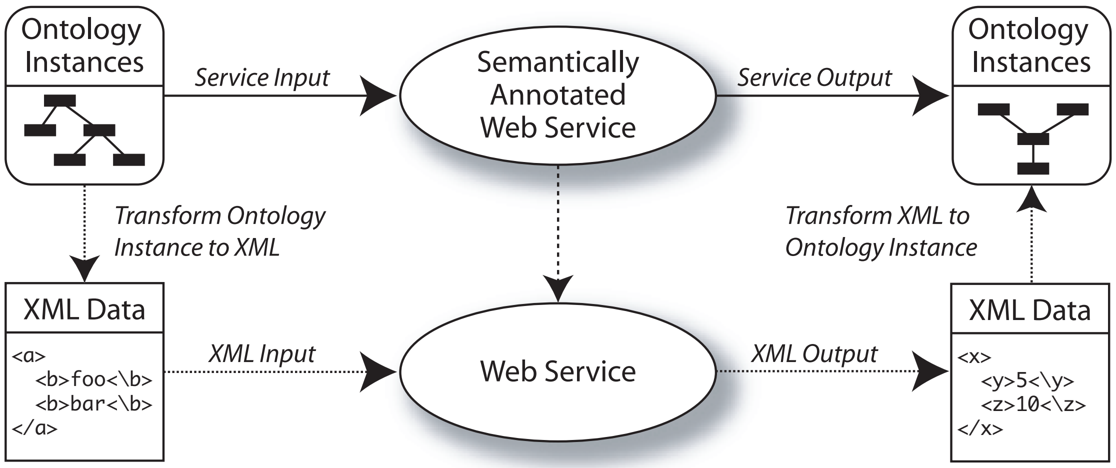


FIGURE 3.5: A semantically annotated Web Service is a traditional Web service that has annotations describing the grounding of ontology concepts to its XML types.

Semantically enabled Web Services are services that consume input and produce output at a conceptual level. We assume these types of service are able to reason on the data received and have a suitable mark-up mechanism to describe their functionality. We illustrate these types of service in Figure 3.6.

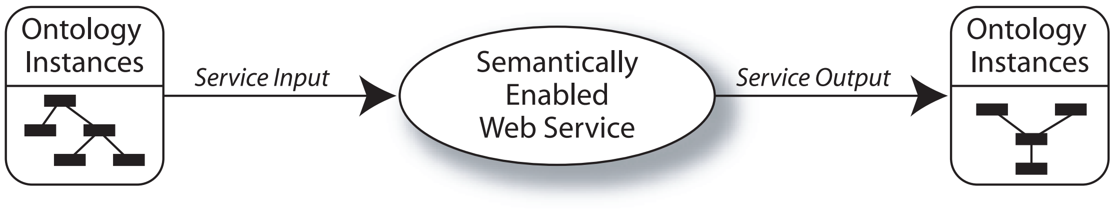


FIGURE 3.6: A Semantically enabled Web Service which consumes input and produces output in the form of ontology instances.

With semantically enabled Web Services, conceptual service definitions are created and maintained by service providers. This restricts compatibility since any potential clients must understand the domain ontologies used by the provider. Given the distributed nature of the Web and the diverse range of communities utilising it, it is likely that several ontologies will develop to explain the same domain using slightly different structures and terms. Service providers must also anticipate the requirements of the client which can be problematic because different clients may use the same service to achieve different goals. For example, the Amazon Web Service (`www.amazon.com`) can be used to purchase CDs, but a client may wish to use the service to find album track listings or cover art. In addition, for a service to be semantically enabled, the provider is forced to provide a semantic description; a complex task which they may not be qualified to perform or wish to spend resources doing so.

A semantically annotated Web Service permits multiple annotations for a single service instance. This allows different organisations and communities to describe Web Services with their own ontologies according to their own interpretations. It also means service providers can still use conventional WSDL documents to expose their capabilities and rely on third party annotations to give them semantics. Finally, by annotating existing definitions rather than altering them, we can ensure compatibility between semantic and non-semantic clients.

The bioinformatics application in which our work is situated is a semantically annotated environment. Services expose their functionality using conventional interface definitions such as WSDL. These interface definitions are then annotated with terms from a bioinformatics ontology, supplying semantics and capturing the meaning of the service.

3.4 Data Integration

The workflow problem we present in our use case emanates from the variety of data formats assumed by service providers. *Data Integration* (the means of gathering information from multiple, heterogeneous sources) also addresses this problem, providing solutions which enable the harvesting of information across differing syntactic representations. Given the similarity of the problem, we investigate the following data integration research: a bioinformatics application that utilises ontologies to capture the meaning of information content; a physics Grid technology that enables transparent access to data ranging over multiple, divergent sources; a geographic dataset integration solution; a semantic data integration system for the web; and a Grid data definition language to support the meaningful interchange of Grid data.

3.4.1 TAMBIS - Data Integration for Bioinformatics

The Transparent Access to Multiple Bioinformatics Information Sources [87] (TAMBIS) framework is designed to support the gathering of information from various data sources through a high-level, conceptually-driven query interface. In this system, information sources are typically proprietary flat-file structures, the outputs of programs, or the product of services, with no structured query interface such as SQL or XQUERY [22], and no standard representation format. A molecular biology ontology, expressed using a description logic, is used in conjunction with functions that specify how every concept is accessed within each data source to deliver an advanced querying interface that supports the retrieval of data from multiple information sources assuming different data representations. The requirements for syntactic mediation are similar to those of data integration: syntactic mediation requires a common way to view and present information from syntactically incongruous sources; data integration systems, such TAMBIS, have achieved this by using conceptual models to describe information source in a way that is independent of representation. While the TAMBIS approach is useful when considering the consolidation of Web Service outputs, it does not support the creation of new data in a concrete format, a process that is required when creating inputs to Web Services.

3.4.2 XDTM - Supporting Transparent Data Access

The need to integrate data from heterogeneous sources has also been addressed by Moreau *et al* [78] within the Grid Physics Network, GriPhyN [50]. Like the bioinformatics domain, data sources used in physics Grids range across a variety of legacy flat file formats. To provide a homogeneous access model to these varying data sources, Moreau *et al* [78] propose a separation between logical and physical file structures. This allows access to data sources to be expressed in terms of the logical structure of the information rather than the way in which it is physically represented. To achieve this, an XML schema is used to express the logical structure of an information source, and mappings are used to relate XML schema elements to their corresponding parts within a physical representation. The XML Data Type and Mapping for Specify Datasets (XDTM) prototype provides an implementation which allows data sources to be navigated using XPATH. This enables users to retrieve and iterate across data stored across multiple, heterogeneous sources. While this approach is useful when amalgamating data from different physical representations, it does not address the problem of data represented using different logical representations. Within a Web Service environment where service are described using WSDL, we can assume homogeneous logical representation because interface types are described using XML schema. Our workflow harmonisation problem arises from the fact that service providers use different logical representations of conceptually equivalent information, i.e. differently organised XML schemas to hold the same conceptual items.

3.4.3 Ontology-based Geographic Data Set Integration

Geographic data comes in a variety of formats: digitised maps, graphs and tables can be used to capture and visualise a range information from precipitation levels to population densities. As new data instances appear, it is important with geographic data sets to recognise their spatial attributes so information can be organised and discovered by regional features such as longitude and latitude, as well as political or geographic location. Uitermark *et al* [92] address the problem of geographic data set integration: the process of establishing relationships between corresponding object instances from disparate, autonomously producing information sources. Their work is situated in the context of update propagation so geographically equivalent data instances from different sources, in different formats, can be identified and viewed as the same instance. Abstraction rules dictate the correspondence between elements from different data models which means the relationship between instances of data in different models can be derived, e.g. they are in the same location or they fall within the same region.

3.4.4 IBIS: Semantic Data Integration

The *Internet-Based Information System* (IBIS) [29] is an architecture for the semantic integration of heterogeneous data sources. A *global-as-view* approach [19, 32] is employed meaning a single view is constructed over disparate information sources by associating each element in a data source to an element in a global schema. A relational model is used as the global schema with non-relational sources wrapped as legacy file formats; Web data and databases models can all be queried using a standard access model. A novel feature of the IBIS architecture is the ability to deal with information gathered via Web forms. This is achieved by exploiting and implementing techniques developed for querying sources with binding patterns [69].

3.4.5 Data Format Definition Language

The Data Format Definition Language (DFDL) [16] is a proposed standard for the description of data formats that intends to facilitate the meaningful interchange of data on the Grid. Rather than trying to impose standardised data formats across vendors, the DFDL language can be used to specify the structure and contents of a file format at an abstract level, with mappings that define how abstract data elements are serialised within the data format. The DFDL API can then be used to parse data and operate over it without regard for the physical representation of the data. This approach has the benefit that information providers can choose to represent their data using the most appropriate format. This is an important consideration for Grid applications because data sets can be large and complex, and therefore, enforcing a particular representation language such as XML is not feasible.

3.4.6 Reflection on Data Integration Techniques

Viewing information sources through a three-tier model [86] allows us to separate different data integration solutions and position our work against them. Figure 3.7

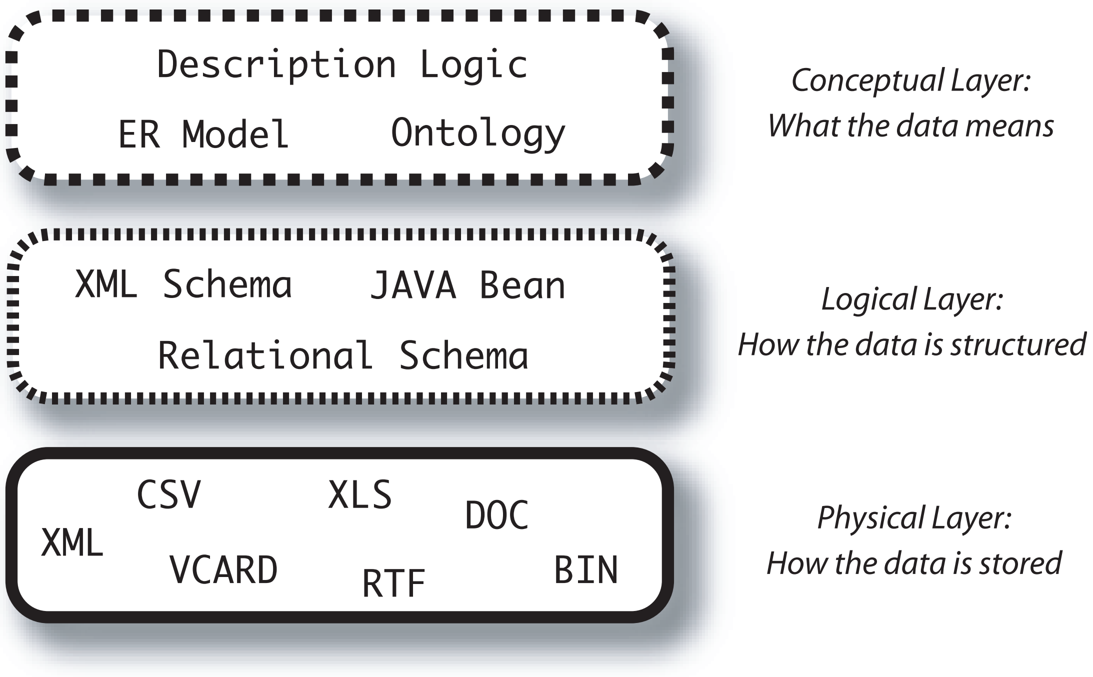

Description Logic
ER Model
Ontology
Conceptual Layer: What the data means
XML Schema
JAVA Bean
Relational Schema
Logical Layer: How the data is structured
CSV
XLS
DOC
XML
VCARD
RTF
BIN
Physical Layer: How the data is stored

Figure 3.7: A Three Tier Model to separate physical storage, logical structure and conceptual meaning of information.

illustrates the relationship between physical representation, logical organisation, and the meaning of data:

1. **Physical Representation - How the file is stored**
   Data can be stored in a variety of formats: proprietary binary files, text files, XML files and databases encompass the most common methods.
2. **Logical Organisation - How the information is structured**
   On top of the physical representation layer, the logical organisation of the data dictates the structure of the information, e.g. XML schema, relational models, etc.
3. **Conceptual - What the data means**
   On top of the logical organisation layer, the conceptual model of an information source specifies what the information means at a high-level of abstraction.

It is common for data integration solutions to use a common representation or uniform access model to facilitate the gathering and processing of information from different representations. In terms of the three-tier model presented in Figure 3.7, a set of heterogeneous formats in one layer can be abstracted in the layer above to support homogeneous data access. For example, different physical file formats can be integrated through a common structural representation, a technique used by DFDL, XDTM and IBIS. If different logical organisations of data exist, a common conceptual model can be used to access data sources through a single view, an approach used by TAMBIS and the integration of geographic datasets. In either case, some form of mappings or wrapper programs are used to translate data. The workflow harmonisation problem that we presented earlier in Chapter 2 stems from the fact that different service providers assume different logical organisations of data (under the assumption that XML schema are used to describe the input and output of Web Services). Therefore, a common conceptual model that describes the contents of different XML schemas can be used to drive the translation of data between different formats. To achieve this, some method is required to assign meaning to XML schema components expressed in a high-level language such as a description logic or ontology. This notion, commonly referred to as XML *semantics*, is discussed in the following section.

3.5 XML Semantics

The idea of assigning semantics (or meaning) to elements and attributes inside XML schemas has been explored in a variety of different ways. In some cases, it is used for data integration purposes; many different XML instances that assume different logical structures are viewed through a common *conceptual* model so queries across different representations and their results are expressed in terms of the meaning of the data that is captured in a high-level model. Kim and Park [64] have developed the XML Meta Model (XMM) to support this kind of functionality. The XMM captures the semantics of XML schemas using a simple *Is-A* relationships: each element and attribute within an XML schema is an instance of a particular concept within the XML meta model.

Schuetzelhofer and Goeschka [85] have employed a set theory approach to assign domain semantics to information represented in XML. A three-layer meta-model graph breaks XML into three levels: (i) the instance-level graph models the existence of elements, attributes, and literals as nodes of a graph and types as their edges (ii) the type-level graph models the XML schema with element and attribute definitions represented as nodes, and type definitions represented as edges (iii) the meta-type-level is comprised of meta-type nodes that model the domain concepts, and meta-type links that represent the relationship between domain concepts. With this three-layer meta model representation of XML, instances of elements in different schemas that share the same meta-type-level nodes are conceptually equivalent. Therefore, a homogeneous view for querying XML data across different logical representations (i.e. different XML schemas) can be achieved through the meta-type level.

Liu *et al* [70] present the XML Semantics Definition Language (XSDL) to support the modelling of XML semantics. Using OWL ontologies to capture the semantics and structure of XML documents, and mappings that declare the relationship between XML schemas and OWL ontologies, different representations of conceptually equivalent information can be viewed through a common ontological model. This approach is also used by An *et al* [7] who define a mapping language to express the relationship between XML DTDs and OWL ontologies.

While these data integration techniques facilitate the viewing and querying of data across different XML representations through a common conceptual model, they do not enable the conversion of data *between* different formats. For workflow harmonisation, when the output format from one service does not match the input format to another service, data needs to be converted from one representation *to* another. To apply data integration techniques that utilise a shared conceptual model of data to the workflow harmonisation problem requires a two-way conversion process: information from one format that is viewed through the conceptual model must be serialised to a different format. This idea has been explored by Balzer and Liebig [14] in the context of Semantic Web Service integration. Again, OWL ontologies are used as a common conceptual model to capture the semantics of XML data structures. Unlike the research presented above, their mapping approach enables the conversion of data from XML to OWL and from OWL to XML providing the mechanism necessary to support workflow harmonisation. However, their mapping language is quite limited: a one-to-one correspondence between

XML elements and OWL concepts is assumed. Through the investigation of real bioinformatics data later in Chapter 5, we find that data structures are not so neat and often the combination of more than one element constitutes a single concept, particular elements can have different semantics depending on their context, and some element's semantics change depending on the values of other elements and attributes.

3.6 Automated Workflow Harmonisation

In this Section, we examine two systems that provide support for automated mediation in service-oriented architectures: a classification based approach where mediator services are used to harmonise data incompatibilities, and an ontology-based approach that generates transformations between syntactically discordant interfaces.

3.6.1 Shim based Service Integration

Hull *et al* [58] have investigated the workflow harmonisation problem within the myGrid project. They dictate that conversion services, or *shims*, can be placed in between services whenever some form of translation is required. They explicitly state that a shim service is *experimentally neutral* in the sense that it has no effect on the result of the experiment. By enumerating the types of shims required in bioinformatics Grids and classifying all instances of shim services, it is hoped that the necessary translation components could be automatically inserted into a workflow or suggested to the user at workflow composition time. However, their work encapsulates a variety of conversion services, not just ones to perform syntactic mediation. Shim services are classified in the following way:

- **Dereferencer**
  In our use case, an accession id is used to retrieve a sequence data record. In bioinformatics services, it is often the case that results from analysis services are references to a record and not the actual record itself. When results from

one service invocation are passed as input to another service, sometimes it is necessary to insert an additional service to retrieve the entire record. This type of intermediate service is classified as a *dereferencer* shim.

- **Syntax Translator**
  When services assume different representation of the same information, a *syntax translator* shim is inserted.

- **Semantic Translator**
  Sometimes a conversion between conceptually similar information is required. For example, a DNA sequence may need to be converted to a protein sequence. This type of service is classified as a *semantic translator* shim.

- **Mapper**
  In our use case, the two sequence retrieval services use the same unique record identifiers (or accession id's). Other bioinformatics services exist to retrieve sequence data, but using different unique identifiers. Services that convert a reference from one system to another are classified as *mapper* shims.

- **Iterator**
  When the output from one service is a set of records, and the input to the next service is a single record, an *iterator* shim can be placed in between services to process each member of the record set individually and combine the results.

For the purpose of this analysis, we consider only the syntax translator shim; the other types of shim service cover integration problems outside the scope of syntactic mediation. The notion that particular types of service can be grouped together (e.g. services for syntax translation) is useful because users can readily identify services that will help them resolve syntactic incompatibilities. From an automation perspective, the classification approach would work if the capability of the conversion service (namely the source type consumed and the destination type produced) can be queried because software components could then find Type Adaptors to meet specific translation requirements at runtime.

3.6.2 Ontology based transformation generation

The SEEK project [24] specifically addresses the problem of heterogeneous data representation in service oriented architectures. Within their framework, each service has a number of ports which expose a given functionality. Each port advertises a *structural type* that represents the format of the data the service is capable of processing. These structural types are specified by references to XML schema types. If the output of one service port is used as input to another service port, it is defined as *structurally valid* when the two types are the same. Each service port can also be allocated a *semantic type* which is defined by a reference to a concept within an ontology. The plugging together of two service ports is *semantically valid* if the output from the first port is subsumed by the input to the second port. Structural types are linked to semantic types by a registration mapping using a custom mapping language based on XPATH. If the plugging together of two ports is semantically valid, but not structurally valid, an XQUERY transformation can be generated to harmonise the two ports, making the link *structurally feasible.* While the SEEK project does present a solution to the problem of harmonising syntactically incompatible services, their work is only compared to the services within the bespoke SEEK framework — the use of specific Web Service technologies such as WSDL or SOAP are not discussed.

3.7 Discovery and Composition

Within any large-scale Grid or Web Services application, the discovery of services and the composition of workflows is a fundamental process. We inspect the technology that facilitates these processes and discuss recent research within these fields.

3.7.1 Grid Based Semantic Service Discovery

Grid environments are not static: new services can appear, services can disappear and existing interfaces can be modified at any time. To cope with this dynamic scenario, service registries are often used to keep track of the services available. Within the myGrid project, the Taverna workbench uses a service registry to maintain a list of available services and presents them to the user. As we mentioned in Section 3.1.2, existing Web Service discovery technologies, such as UDDI, are only able to provide primitive discovery mechanisms based on simple string matching of service descriptions and classifications of service instances. Next generation discovery components, such as FETA [71], supply more advanced service location mechanics by exploiting semantic service annotations. To support this kind of discovery, the service registry has evolved into a more complex component. Instead of simply storing interface definitions, it is now necessary for Grid registries to support the annotation of service interfaces with additional semantics by both service providers and third parties. To enable semantic service discovery, query interfaces must be provided to support the searching and retrieval of services in terms of the service semantics.

The Grimoires service registry [93] is an example of such a next generation Grid registry, supporting advanced service annotation and discovery. Grimoires works on top of existing Web Service standards providing annotation support for WSDL service definitions and UDDI service records. Meta-data is stored using RDF [66] triples and a query interface is provided using RDQL. The meta-data attachment policy is generic so it can support a range of annotation approaches such as OWL-S and WSDL-S.

3.7.2 Web Service Composition with Semantics

In scientific Grid applications, such as Taverna, workflows are used to capture the experimentation process. With the introduction of semantic service annotations, conventional workflow models can be augmented in two ways:

1. **Abstract Workflow Definitions**
   Traditional workflows are specified over service instances. For example, the workflow we present in our use case is specified over the DDBJ-XML or XEMBL sequence retrieval services and the NCBI-Blast service. Given that semantic annotations describe service interfaces at conceptual level, new

workflow models are being formulated to allow users to specify service compositions at an abstract level [38, 73]. This allows users to express their desired workflow in terms of the kind of service used, rather than the actual service instances. Our use case scenario could be expressed as a sequence retrieval service which gets a record and passes it to an alignment service. This abstract workflow definition can then be mapped onto an enactable workflow instance using a semantic service discovery component [95].

2. **Automated Composition**
   By utilising planning techniques from the artificial intelligence community, it is possible to generate a service composition that achieves a high-level goal that is not achievable by a single service instance [18, 21]. For example, our use case workflow could be specified as a single task that consumes a sequence accession id and produces sequence alignment results.

In both of these scenarios, we find that services may be plugged together because their semantic descriptions deem them compatible. However, as we have shown in Chapter 2, semantically interoperable services (services that share the same semantic types) are not necessarily syntactically compatible. Therefore, automated workflow harmonisation is critical to the success of these applications.

3.8 Conclusions and Analysis

We have shown that the application of Semantic Web technology to Web Services can facilitate more advanced service discovery and autonomous invocation. By using a bioinformatics ontology, such as the one presented by Wroe *et al* [94], a Web Service's characteristics can be defined using high-level terminology from a shared conceptualisation that is familiar to users. To support this type of annotation, next generation service registries, such as GRIMOIRES, can be used in combination with advanced discovery components, such as FETA, to supply a rich Web Service environment that supports users in the composition of workflows and facilitates the scientific process.

In Chapter 2, Section 2.4, we identified that workflow composition is hindered by the fact that service providers and consumers often assume different representations for conceptually equivalent information. While existing manual solutions do provide the necessary syntactic mediation for workflow harmonisation, an automated solution is preferable for two main reasons:

1. The Semantic Web Service philosophy is centered around the notion that users should be able to coordinate the exchange and processing of information between parties using high-level terms from shared conceptualisations without concern for the interoperability issues. When users are forced to consider the data formats assumed by providers and consumers and how they relate to each other, this basic premise is violated.

2. A considerable amount of effort is required for users to insert the appropriate mediation components. While it may be the case that a Type Adaptor exists to harmonise a particular dataflow between services, users are unable to share and discover them, so duplication is rife. Furthermore, users in these types of domain, such as bioinformatics, are not experts in Web Service composition so the harmonisation of dataflow is a daunting and complex task.

Through our investigation in related work, we identified that previous data integration work also tackles to problem of heterogeneous data representation; projects such as TAMBIS and SEEK have successfully used shared conceptual models to provide a homogeneous access model across disparate information formats. This type of approach can be implemented using a mapping language that provides a data representation with clear semantics, allowing pieces of information in different formats to be identified through a common term or concept. However, existing approaches that apply semantics to XML data structures to drive homogeneous data access are usually one way processes: information can be gleaned from different formats and viewed through a common model, but it cannot be converted between different syntactic models. The workflow harmonisation problem we tackle requires the conversion of data between formats, and therefore, further work is required.

In the next Chapter, we present the WS-HARMONY architecture: a framework to support automated workflow harmonisation. Data integration techniques that rely on common conceptualisations to capture the structure and semantics of different data formats are used, along with Web Service discovery technology, to facilitate the automatic discovery and inclusion of the appropriate Type Adaptors at runtime.

Chapter 4

WS-HARMONY: An Architecture for Automated Workflow Harmonisation

In Chapter 2, we presented the problem that occurs in semantically-annotated Web Service environments when a service provider and a service consumer assume different syntactic representations for conceptually equivalent information. While this motivating use case highlights the impact of this problem when users are composing services based on their semantic definitions (e.g. high-level conceptualisations of the service inputs and outputs), we can also imagine similar problems arising when automatic planning techniques are used to generate workflows.

As indicated in Chapter 3, Section 3.7, much research has been undertaken to convert abstract workflows to concrete specifications [38, 73], as well as the generation of workflows to fulfill tasks not achievable by a single service [18, 21]. In both of these examples, services may be joined by a planning algorithm because they are deemed *semantically compatible* (through inspection of the service's semantic annotations). Since the planing techniques listed above do not consider low-level interoperability issues when joining services, they may generate workflows that cannot be reliably invoked. Therefore, the harmonisation solution presented in this dissertation is important not only for the development of applications like Taverna that provide users with an interaction to semantically-annotated Web

Services, but also to the Semantic Web research field as a whole: there are many situations when services are connected because they *should* fit, even though they may not agree on the same syntactic model.

In this Chapter, we present the Web Service Harmonisation Architecture (WS-HARMONY) that supports the invocation of Web Services and automatic reconciliation of data formats whenever semantically compatible but syntactically incongruous service interfaces are joined. For this architecture, we do not believe it necessary to conform to a particular Semantic Web Service annotation model (such as OWL-S, WSDL-S or FETA) because the same harmonisation problem can arise when using any of them, and the same solution can be employed. The assumptions we make are that all service interfaces are defined using WSDL with their respective message parts specified using XML schema. This makes our solution compatible with any technology that conforms to these widely used standards, including user-oriented applications and automatic composition software.

Broadly, the architecture can be split into two sections: the syntactic mediation approaches supported and the infrastructure created to enable them; and the discovery of Type Adaptors to automate the process of syntactic mediation and the invocation of target services. The contributions of this Chapter are:

1. **Scalable mediation approach**
   To support the translation of data between different formats, we make use of shared conceptual models expressed with the ontology language OWL. Individual data formats are mapped to a conceptual model using a declarative and composable mapping language so conceptually equivalent elements within different representations are linked via a common concept. This approach provides better scalability as the number of compatible data formats increases than directly translating data between formats.

2. **Type Adaptor generation from mapping rules**
   By consuming mappings that specify the meaning of XML schema components through a shared conceptual model, our Configurable Mediator is able to masquerade as a bespoke Type Adaptor on demand to fulfill a given translation requirement.

3. **Middleware to facilitate the sharing and automatic discovery of Type Adaptors**
   One novel aspect of the WS-HARMONY architecture is the use of WSDL to describe Type Adaptors and mapping specifications. This enables us to reuse existing registry technology to enable the sharing and discovery of adaptor components.

The first half of this Chapter is concerned with mediation, beginning in Section 4.1 with a classification of mediation approaches and how OWL can be used to drive data translation with an intermediate representation. Section 4.2 shows how mediation components fit into existing workflow execution frameworks and introduces the Configurable Mediator: a software component that translates data using mappings specified between XML schemas and OWL ontologies. Section 4.3 discusses the requirements of our mediation approach and argues for the de-coupling of translation specifications from service descriptions, as well as highlighting the benefits of a modular language to describe translation. The latter half of the Chapter shows how we automate the process of syntactic mediation, starting in Section 4.4 with an overview of our advertising and discovery techniques before the presentation of the WS-HARMONY architecture as a whole in Section 4.5.

The WS-HARMONY architecture presented in this Chapter is given at a high level: many of the technical aspects are reserved for later Chapters where they are presented in more detail. References to more detailed explanations are given at the appropriate place, as well a summary of contributions at the end in Section 4.6.

4.1 Mediation Approach

The conversion of data between different formats in large-scale and open systems, such as the Grid and Web Services, can be separated into two approaches:

1. **Direct Mediation**
   When many different formats exist to represent conceptually equivalent information, Type Adaptors may convert data between formats directly. We

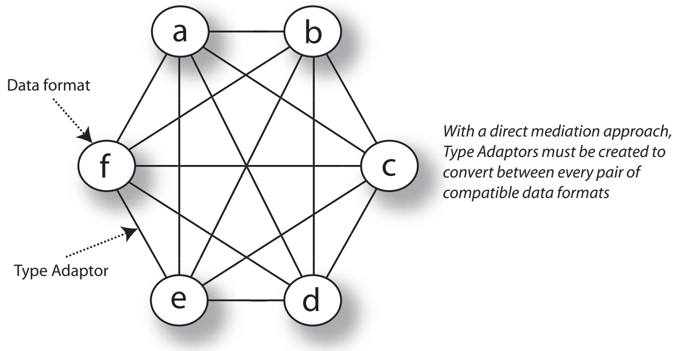

a
b
c
d
e
f
Data format
Type Adaptor
With a direct mediation approach, Type Adaptors must be created to convert between every pair of compatible data formats

FIGURE 4.1: Direct Mediation: Converting data directly between formats

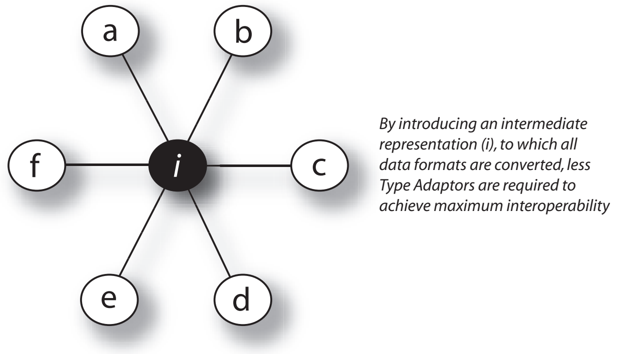

a
b
c
d
e
f
i
By introducing an intermediate representation (i), to which all data formats are converted, less Type Adaptors are required to achieve maximum interoperability

FIGURE 4.2: Intermediary Mediation: Converting data to an intermediate representation

illustrate this approach in Figure 4.1 where we show six compatible data formats (a to f) and the number of Type Adaptors required (connecting lines). As the number of compatible data formats increases, the number of Type Adaptors required is $O(n^2)$. Whenever a new format is introduced, $O(n)$ Type Adaptors must be created (one for each existing format) to ensure maximum interoperability.

2. **Intermediary-based Mediation**
   By introducing an intermediate representation to which all data formats are translated (Figure 4.2), the number of Type Adaptors required is $O(n)$ as the number of compatible formats increases. Also, when a new data format is conceived, only one Type Adaptor is required to convert this new format to the intermediate representation.

Current systems (such as the Taverna application discussed in Chapter 2) employ a direct mediation approach: conversion components translate data directly from one format to another. Given the poor scalability of this approach and the large overhead when introducing new formats, the WS-HARMONY architecture is centered around the use of an intermediate representation.

While it is outside the scope of this dissertation, a natural progression for this view of data translation is to consider the impact of multiple intermediate representations. We can imagine that a single ontology is constructed to capture the structure and semantics of some data (such as a sequence data record in our use case) to provide a particular application with a single view over heterogeneous formats. However, it is likely that different ontologies would be developed for the same data source because other communities will have different interpretations of the data structure and the terms used. To illustrate this idea, Figure 4.3 shows three different intermediate representations (i_1, i_2 and i_3) and a number of different data formats (including x and y). If a transformation exists between i_1 and i_3, x has a transformation to i_1, and y has a transformation from i_3, then an item in format x may be converted to format y via the intermediate representations i_1 and i_2.

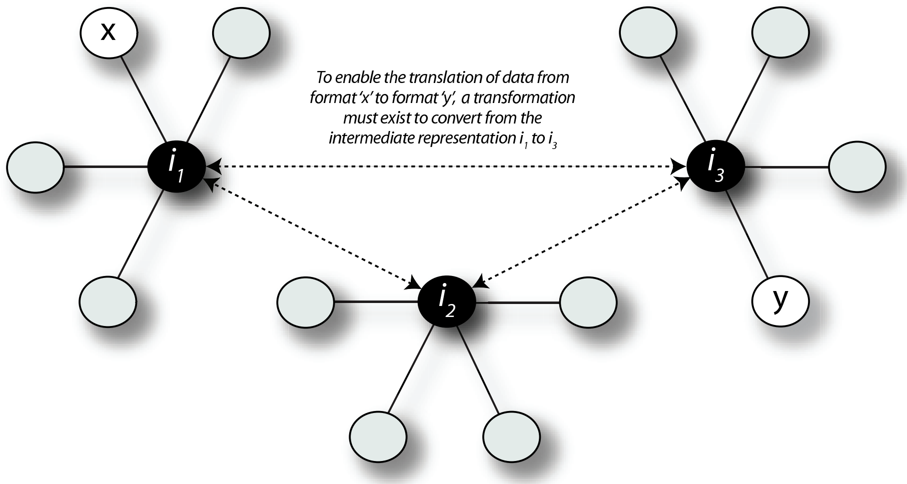

x
To enable the translation of data from format 'x' to format 'y', a transformation must exist to convert from the intermediate representation i_1 to i_3
i_1
i_3
i_2
y

Figure 4.3: Joining of Intermediate representations

4.1.1 Using OWL as an Intermediate Representation

The Web Ontology Language (OWL) [83] is an ontology specification language that enables the publishing and sharing of conceptual models on the Web. By extending the existing mark-up capabilities of RDF and RDFS and combining reasoning capabilities from the description logic community [57, 54], OWL embraces the name-spacing and linking benefits of the Web to support sharing and reuse, and provides the necessary language constructs to model complex domain knowledge. Using OWL to capture the structure and semantics of XML data has been reviewed in Chapter 3, Section 3.5 and is a proven data integration technique. To illustrate this idea against our bioinformatics use case, we present an ontology to describe Sequence Data Records in Figure 4.4. Complete OWL listings for this ontology can be found in Appendix A, Listing A.1.

The main concept, *Sequence_Data_Record* (centre of Figure 4.4), has the datatype properties *accession_id* (denoting the unique id of the dataset) and *description* (a free-text annotation). Each sequence data record has a *Sequence* that contains the string of sequence *data*, the *length* of the record and its *type*[1]. A sequence data record contains a number of *References* that point to publications that describe the particular gene or protein. Each *reference* has a list of *authors*, the *journal* name, and the paper publication *title*. Sequence data records also have a number of different features, each having a *Feature_Location* that contains the *start* and *end* position of the feature. There are many different

[1]Type here does not denote a syntactic type - it is a kind of sequence.

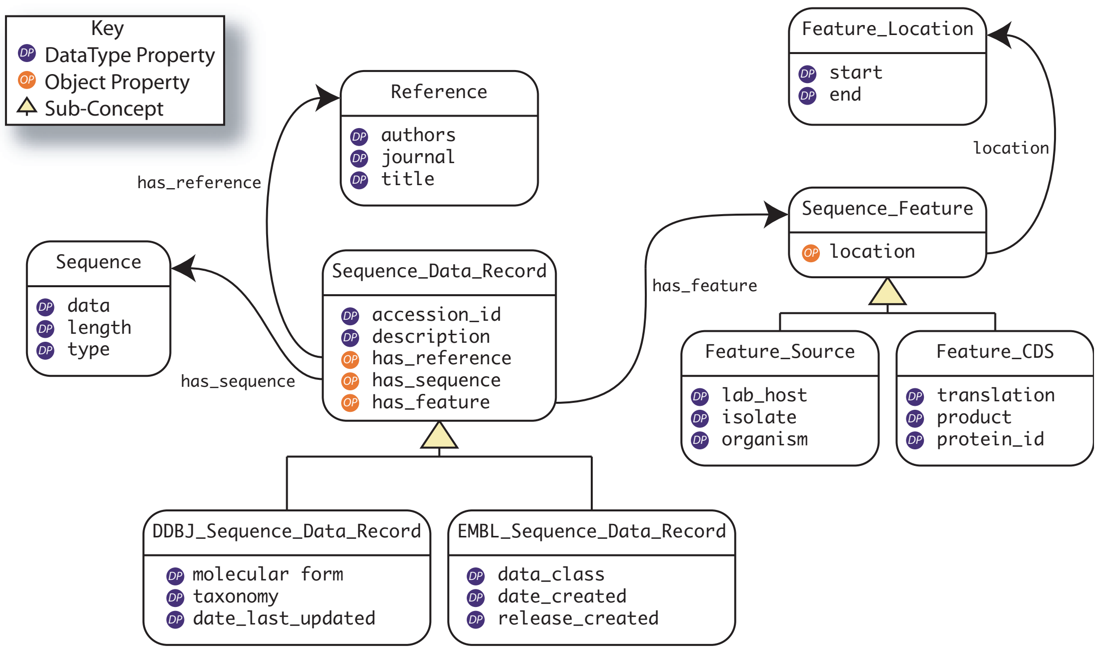


FIGURE 4.4: An Ontology to describe sequence data records

types of feature; we show two common ones in this example: *Feature_Source* and *Feature_CDS*. Both of these concepts are sub-classes of the *Sequence_Feature* concept which means they inherit properties assigned to the parent class. In the case of a sequence feature, they all contain a location, but each has its own list of properties: *lab_host*, *isolate* and *organism* are properties of the *Feature_Source* class; and *translation*, *product* and *protein_id* are properties of the *Feature_CDS* class. The *Sequence_Data_Record* concept also has two sub-concepts: *DDBJ_Sequence_Data_Record* and *EMBL_Sequence_Data_record*. These classes capture the fact that while both the DDBJ and XEMBL formats contain mainly the same information, there are attributes of each record that are unique to their format. For example, repository-specific information such as the date created or date last updated.

Fragments of XML describing a sequence feature in both DDBJ and EMBL formats are given in Listing 4.1 and 4.2 These two representations essentially contain the same information in different formats: The Feature type is CDS, it has a product, protein_id, translation and location. Figure 4.5 gives a visual representation of the concept instances that would be used to capture this sequence feature information. An instance of the *Feature_CDS* class would be used with three datatype properties holding the translation, product and protein_id. The feature location

```
<feature name="CDS">
  <qualifier name="product">capsid protein 2</qualifier>
  <qualifier name="protein_id">BAA19020.1</qualifier>
  <qualifier name="translation">MSDGAVQPDGGQPAVR...</qualifier>
  <location type="single" complement="false">
    <locationElement type="range" accession="AB000059">
      <basePosition type="simple">1</basePosition>
      <basePosition type="simple">1755</basePosition>
    </locationElement>
  </location>
</feature>
```

LISTING 4.1: Sample XML from a EMBL formatted Sequence Data Record

```
<FEATURES>
   <cds>
      <location>1..1755</location>
      <qualifiers name="product">capsid protein 2</qualifiers>
      <qualifiers name="protein_id">BAA19020.1</qualifiers>
      <qualifiers name="translation">MSDGAVQPDGGQPAVR...</qualifiers>
   </cds>
</FEATURES>
```

LISTING 4.2: Sample XML from a DDBJ-XML formatted Sequence Data Record

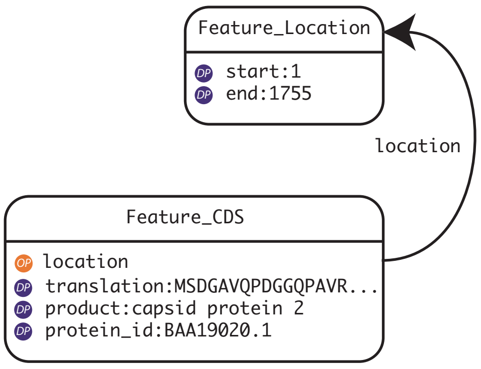


FIGURE 4.5: An OWL concept instance to describe a feature from a Sequence Data Record

information would be represented using an instance of the *Feature_Location* class and would be linked to the *Feature_CDS* via the object property location. With a common ontology in place to describe bioinformatics data, syntactically incongruous dataflow between two services operating in this domain can be harmonised by translating data from one representation to another via the intermediate OWL model. This idea is exemplified in Figure 4.6 against our bioinformatics use case from Chapter 2. In this example, the output from the DDBJ-XML service is converted to its corresponding concept instance (the *Sequence_Data_Record* concept), which can in turn be converted to FASTA format for input to the NCBI-Blast service. We define two terms to distinguish between these conversion processes:

- **Conceptual Realisation**

  The conversion of an XML document to an OWL concept instance.

- **Conceptual Serialisation**

  The conversion of an OWL concept instance to an XML document.

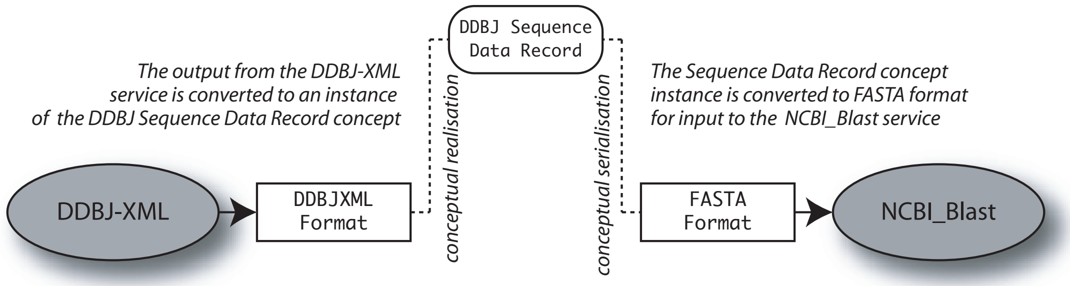

DDBJ Sequence Data Record
The output from the DDBJ-XML service is converted to an instance of the DDBJ Sequence Data Record concept
The Sequence Data Record concept instance is converted to FASTA format for input to the NCBI_Blast service
conceptual realisation
conceptual serialisation
DDBJ-XML
DDBJXML Format
FASTA Format
NCBI_Blast

FIGURE 4.6: Using an ontology instance as a mediator to harmonise services with incompatible interfaces.

To facilitate these conversion processes, we assume a canonical representation for OWL concept instances. This allows us to view conceptual realisation and conceptual serialisation as XML to XML transformations. While it is common for OWL users to specify OWL concept instances using RDF/ XML syntax, XML schemas do not usually exist to validate them. Therefore we automatically generate schemas using the OWL-$\mathcal{XIS}$ generator, presented in full in Chapter 7, Section 7.2.

4.1.2 Mapping XML to OWL and vice versa

To enable the transformation of XML data to OWL and vice versa, we present the declarative mapping language FXML-M (formalised XML mapping). FXML-M is modular and composable to embrace XML schema reuse meaning XML schema components[2] can be mapped individually to OWL concepts and properties. We formalise this mapping language and the transformation process in Chapter 5 after deriving requirements from real bioinformatics data sets. FXML-M is designed to accommodate complex relationships: collections of XML components can be mapped to single elements (and vice versa); components can be mapped differently based on the existence and values of other elements and attributes; components can be mapped depending on their context within an XML document, and some basic string manipulation support is offered through the use of regular expressions. To this end, FXML-M provides a set of novel language constructs that do not exist in other approaches [64, 85, 70, 7, 14].

An implementation of the FXML-M language is provided through a SCHEME [63] library called FXML-T, presented in Chapter 6. Through empirical testing, we show that our implementation scales well with respect to increasing document and schema sizes, offers composability with almost zero cost, and is efficient compared to other translation implementations when used with bioinformatics data sets.

[2]We use the term components to refer to XML elements, attributes and literal values

4.2 Mediation Infrastructure

In Section 4.1, we specified two mediation approaches: direct and intermediary-based. Although direct mediation does not scale well, current Grid and Web Services infrastructures already expose this functionality. Therefore, the WS-HARMONY architecture is designed to cope with both approaches. To present our architecture, and position our contribution against existing technology, the following sub-sections show current workflow invocation models, how they are affected with a direct mediation approach, and what changes intermediary-based mediation requires.

4.2.1 Conventional Workflow Invocation

Since we are augmenting an existing Grid infrastructure, we begin by showing the current topology in Figure 4.7. When executing workflows in a service-oriented environment, a Workflow Enactment Engine, such as FreeFluo [46] or activeBPEL [3], is used to control the invocation of Web Services. The Workflow Enactment Engine takes a workflow specification document describing the services to invoke, the order in which to invoke them, the dataflow between services, and optionally

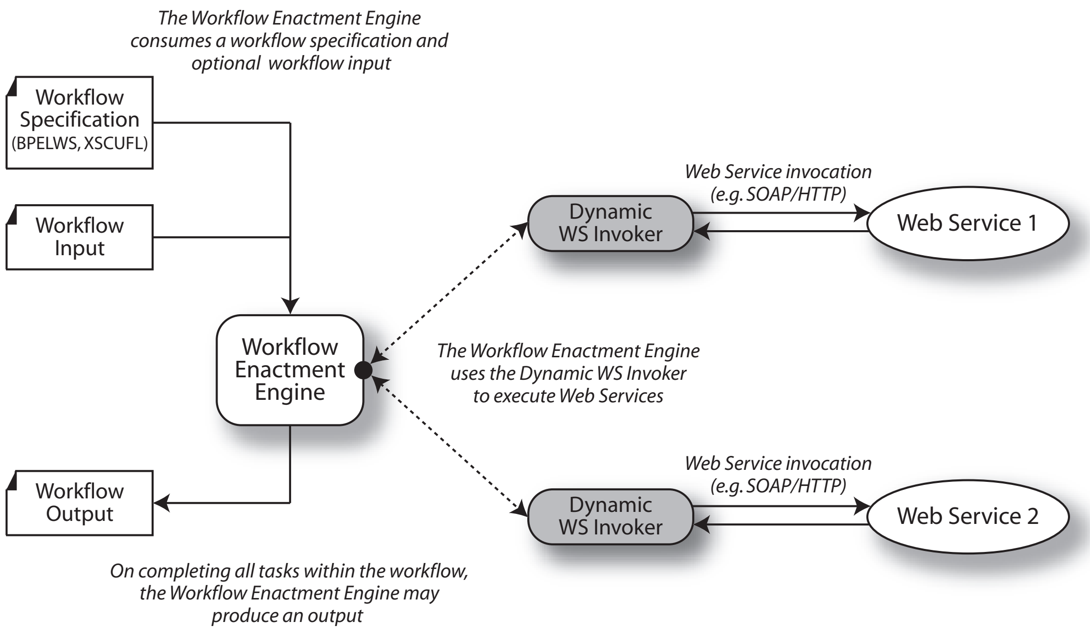

The Workflow Enactment Engine consumes a workflow specification and optional workflow input
Workflow Specification (BPELWS, XSCUFL)
Workflow Input
Workflow Enactment Engine
Workflow Output
The Workflow Enactment Engine uses the Dynamic WS Invoker to execute Web Services
Dynamic WS Invoker
Web Service invocation (e.g. SOAP/HTTP)
Web Service 1
Dynamic WS Invoker
Web Service invocation (e.g. SOAP/HTTP)
Web Service 2
On completing all tasks within the workflow, the Workflow Enactment Engine may produce an output

FIGURE 4.7: Current Invocation Framework for Workflow based applications

some workflow inputs. To support the invocation of arbitrary Web Services, the WS-HARMONY architecture includes a Dynamic Web Service Invoker (DWSI). While current Web Service invocation technologies, such as Apache Axis [10], are adequate in static environments where service definitions are known at design / compilation time, they do not cater well for the invocation of previously unseen services. The DWSI is able to call arbitrary WSDL [33] defined services that bind with SOAP [52] encoding over HTTP transport. This part of the WS-HARMONY architecture is presented in full later in Chapter 7 where a performance evaluation against Apache Axis shows that the DWSI has a lower invocation overhead.

4.2.2 Direct Mediation Workflow Harmonisation

To cater for any syntactically incompatible services, extra stages must be inserted into the workflow to perform syntactic mediation. Figure 4.8 shows various kinds of Type Adaptor (an XSLT script, JAVA class, or Web Service invocation) that could be used as a direct mediator to harmonise the data incompatibility. Current solutions require the manual discovery and insertion of adaptor components into the workflow specification, and thus the workflow designer must consider low-level interoperability issues.

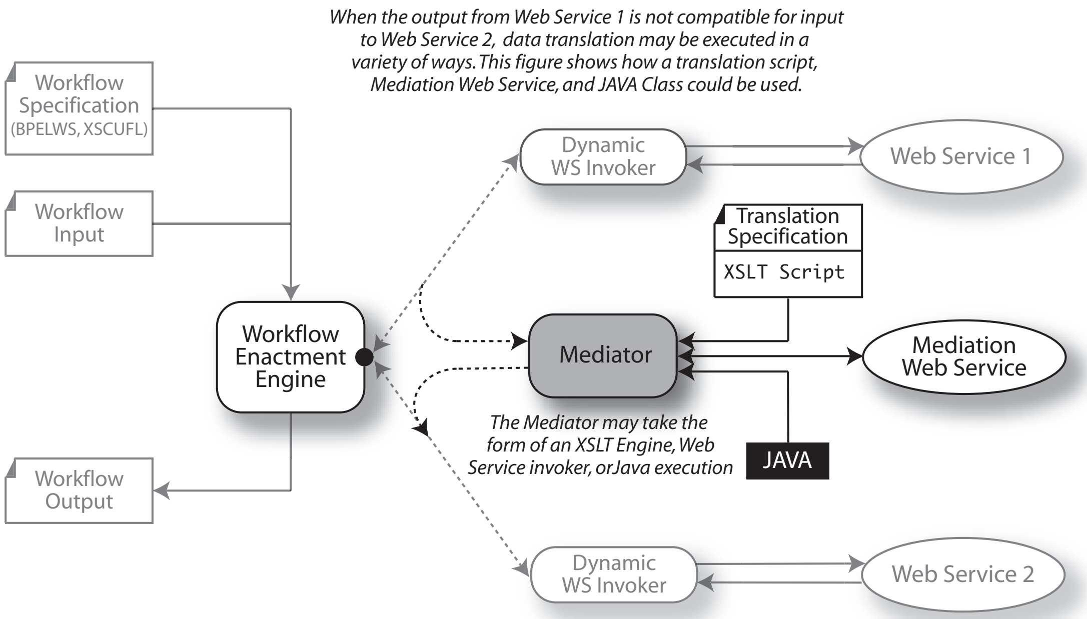

When the output from Web Service 1 is not compatible for input to Web Service 2, data translation may be executed in a variety of ways. This figure shows how a translation script, Mediation Web Service, and JAVA Class could be used.
Workflow Specification (BPELWS, XSCUFL)
Workflow Input
Workflow Enactment Engine
Workflow Output
Dynamic WS Invoker
Web Service 1
Translation Specification
XSLT Script
Mediator
Mediation Web Service
The Mediator may take the form of an XSLT Engine, Web Service invoker, orJava execution
JAVA
Dynamic WS Invoker
Web Service 2

FIGURE 4.8: Syntactic Mediation in the context of workflow

4.2.3 Intermediary-based Workflow Harmonisation

In order to translate XML data from one format to another via an intermediate representation in OWL, as we described in Section 4.1.1, two translation specifications are required: one for conceptual realisation and one for conceptual serialisation (Figure 4.9). The WS-HARMONY architecture supports on-the-fly creation of Type Adaptors using mapping specifications in FXML-M through the Configurable Mediator (C-MEDIATOR) component. The C-MEDIATOR, pictured in detail in Figure 4.10, consumes a serialisation and realisation specification (expressed using FXML-M) and uses them to transform an XML document in one format to an XML document in a different format via an intermediate OWL concept instance. The Translation Engine, built using FXML-T, transforms XML data by consuming mapping rules expressed in FXML-M, and JENA is used to hold the intermediate OWL model. A full breakdown of the C-MEDIATOR and Translation Engine are provided in Chapter 6.

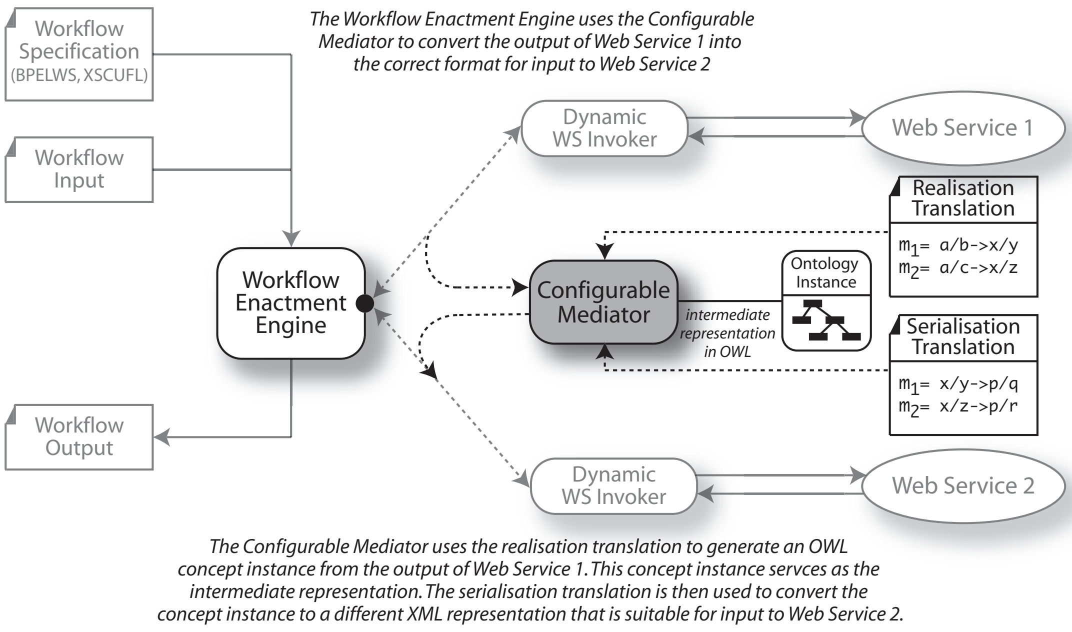

Workflow Specification (BPELWS, XSCUFL)
Workflow Input
Workflow Output
Workflow Enactment Engine
The Workflow Enactment Engine uses the Configurable Mediator to convert the output of Web Service 1 into the correct format for input to Web Service 2
Dynamic WS Invoker
Web Service 1
Realisation Translation
m1= a/b->x/y
m2= a/c->x/z
Configurable Mediator
Ontology Instance
intermediate representation in OWL
Serialisation Translation
m1= x/y->p/q
m2= x/z->p/r
Dynamic WS Invoker
Web Service 2
The Configurable Mediator uses the realisation translation to generate an OWL concept instance from the output of Web Service 1. This concept instance servces as the intermediate representation. The serialisation translation is then used to convert the concept instance to a different XML representation that is suitable for input to Web Service 2.

FIGURE 4.9: Modified Invocation Framework featuring a configurable mediator

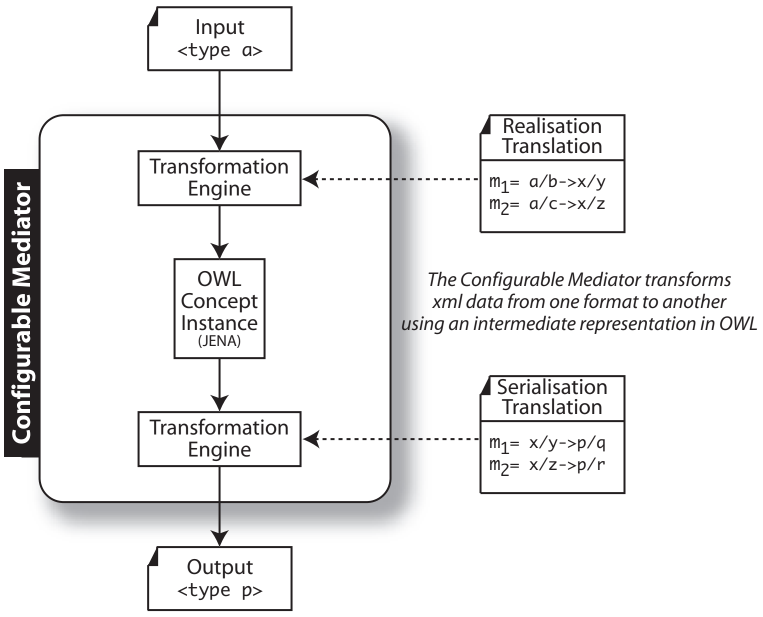

Input
<type a>
Configurable Mediator
Transformation Engine
OWL Concept Instance (JENA)
Transformation Engine
Realisation Translation
m1= a/b->x/y
m2= a/c->x/z
The Configurable Mediator transforms xml data from one format to another using an intermediate representation in OWL
Serialisation Translation
m1= x/y->p/q
m2= x/z->p/r
Output
<type p>

FIGURE 4.10: A high-level view of the Configurable Mediator

4.3 Mediation Specification Requirements

The specification of mappings between XML and OWL is central to our workflow harmonisation solution since they provide the mechanisms necessary to perform syntactic mediation. At a fundamental level, we have split the mediation process into two translation operations: conceptual serialisation, the process of converting XML to OWL, and conceptual realisation, the process of converting OWL to XML. A single Web Service may offer a number of operations: for example, the DDBJ-XML Service we use in our use case offers many operations over sequence data records. As we highlighted in Chapter 2, Section 2.5, XML schema definitions are often reused when services offer multiple operations over the same, or subsets of the same data. A simple service annotation approach defines translations for each Web Service operation. This is the approach taken by OWL-S where XSLT scripts are used to define the transformation for each operation input and output. If this technique is used to annotate the DDBJ-XML service, separate annotations would be needed to describe the “Get SWISS record”, “Get EMBL record” and “Get UNIPROT record” operations. Furthermore, when we consider the “Get Sequence Features” operation, we see that a subset of the same transformation is required because the output is a subset of the full sequence data record. This annotation approach has two major limitations:

1. **Close coupling of mapping specification**
   For the DDBJ-XML service, which offers operations over exactly the same data types, it would be better to de-couple the mapping specifications from the service description for two reasons: (a) the same mappings could be reused by each operation resulting in less work during the annotation process, (b) if the data format and its corresponding ontology definition are modified, only one change to the mapping would be required. The DDBJ-XML service offers 60 different operations to retrieve sequence data records so de-coupling is an important consideration.

2. **No support for mapping reuse**
   As we illustrated earlier in Figure 2.7, the DDBJ-XML service provides operations that return subsets of complete sequence data records. Rather than use separate mappings to describe how each possible subset of the sequence data record is translated to and from an OWL concept instance, it is better to describe how each *part* of the sequence data record is translated using a declarative language, in effect, providing building blocks to construct Type Adaptors.

Therefore, the WS-HARMONY architecture offers a scalable mediation solution, both in terms of the mediation approach (which is based on an intermediate representation), and in the way mappings are de-coupled from the interface definition.

4.4 Discovery of Type Adaptors

The mediation infrastructure presented in Section 4.2 assumes that all Type Adaptor components, either for direct mediation or through an intermediate OWL representation, are known. To enable *automated mediation*, i.e. discover the appropriate translation components without human direction, WS-HARMONY makes use of a registry that stores Type Adaptor information. Since Type Adaptors come in many forms, e.g. application code, scripts, mapping specifications and Web Services, we separate their definitions into abstract capability (what the input and output types are) and concrete implementation (how the Type Adaptor is invoked). Under this assumption, all Type Adaptors can be described using WSDL, and retrieved according to their input and output types. Because WSDL is used to define Type Adaptor functionality, existing Web Service registry technology can be reused. WS-HARMONY relies on the GRIMOIRES [93] registry to support the advertising, sharing and discovery of WSDL Type Adaptor definitions, as we illustrate in Figure 4.11. This part of the WS-HARMONY architecture is presented in full in Chapter 7 where a full explanation of WSDL and the GRIMOIRES registry is given.

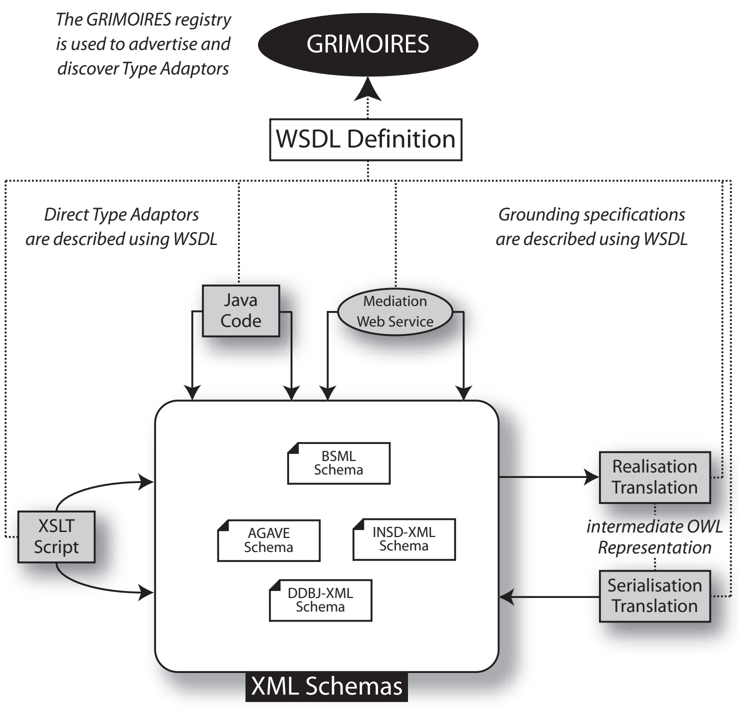

The GRIMOIRES registry is used to advertise and discover Type Adaptors
GRIMOIRES
WSDL Definition
Direct Type Adaptors are described using WSDL
Grounding specifications are described using WSDL
Java Code
Mediation Web Service
BSML Schema
AGAVE Schema
INSD-XML Schema
DDBJ-XML Schema
XSLT Script
Realisation Translation
intermediate OWL Representation
Serialisation Translation
XML Schemas

FIGURE 4.11: WSDL is used to describe Type Adaptors which are registered with GRIMOIRES

4.5 Automated Workflow Harmonisation

With a mediation infrastructure in place that supports the translation of data using direct and intermediary mediators, and a registry containing mediator descriptions, the complete WS-HARMONY architecture can be viewed in terms of Web Services, XML schemas, OWL ontologies, and the GRIMOIRES registry, as we show in Figure 4.12:

- **Web Services**
  Web Services (bottom left of Figure 4.12) are described using WSDL by the service provider. The syntactic type of any operation input or output is defined by a reference to an XML schema type or element.

- **XML Schemas**
  XML schemas (bottom right) are created by service providers to describe the datasets consumed and produced by their Web Services. Direct Mediators (e.g. XSLT scripts, Web Services and bespoke programs) may translate data directly between formats. Mappings supply the necessary translation specification to perform conceptual serialisation and conceptual realisation and enable intermediary-based mediation. In effect, this allows XML data to be turned to and from an OWL concept instance.

- **Ontologies**
  Ontologies (top) capture the structure and semantics of the XML data formats and provide the semantic types for service inputs and outputs necessary for semantic service discovery.

- **Registry**
  The service registry (centre) is used to store WSDL interfaces for Web Services and their corresponding semantic annotations. Any Type Adaptors (both direct and intermediary) are also described using WSDL so the existing Grimoires query interface can be used for discovery according to the required input and output types.

4.6 Conclusions and Contribution Summary

To supply the invocation and mediation framework presented in this Chapter, we make three distinct contributions that are presented in detail in the following Chapters:

1. **A Modular Transformation Theory**
   As we stated earlier in Section 4.3, a good transformation approach for conceptual serialisation and conceptual realisation is modular. On investigating

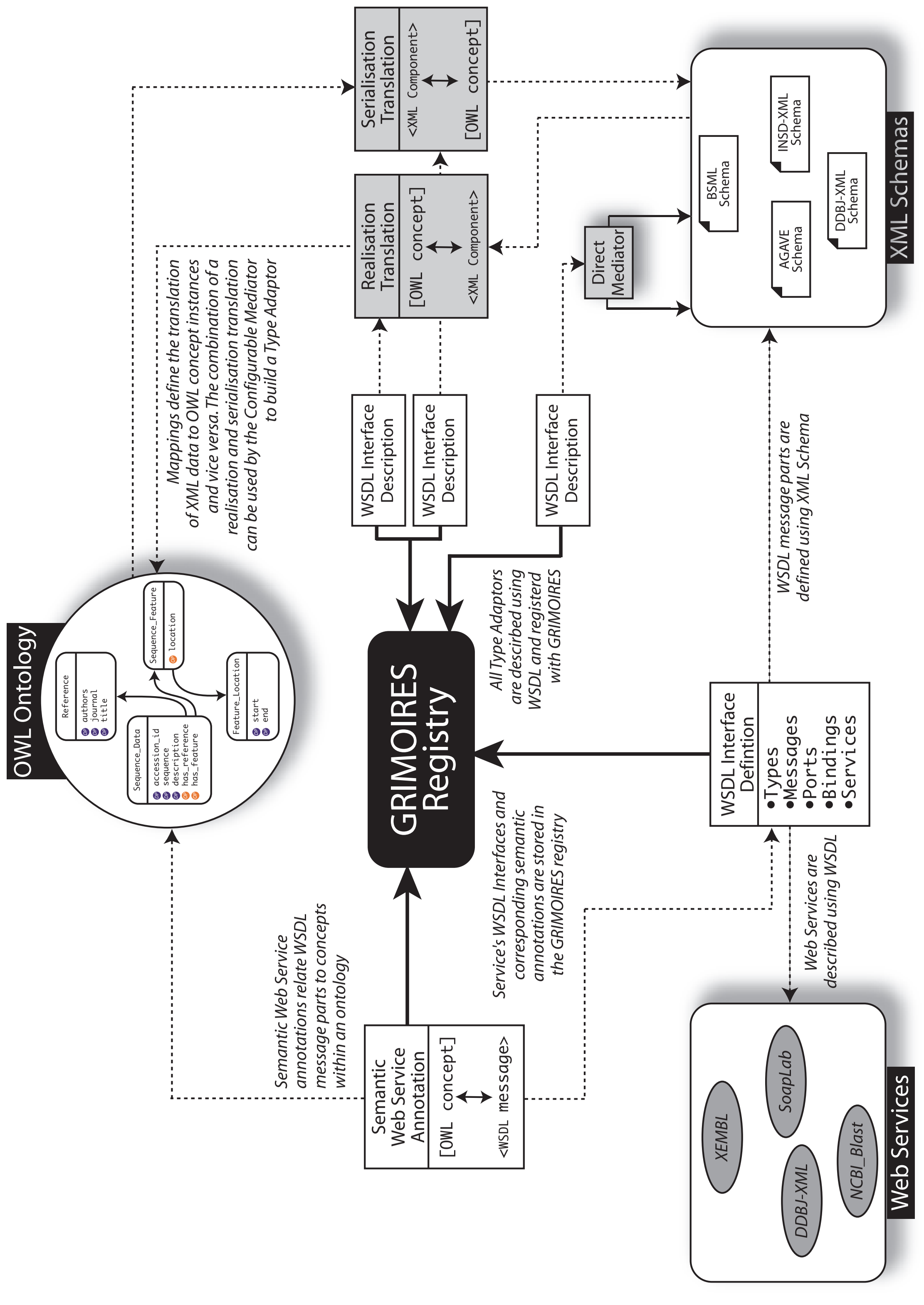

OWL Ontology
Reference
authors
journal
title
Sequence_Data
accession_id
sequence
description
has_reference
has_feature
Sequence_Feature
location
Feature_Location
start
end
Mappings define the translation of XML data to OWL concept instances and vice versa. The combination of a realisation and serialisation translation can be used by the Configurable Mediator to build a Type Adaptor
Realisation Translation
[OWL concept]
<XML Component>
Serialisation Translation
<XML Component>
[OWL concept]
WSDL Interface Description
WSDL Interface Description
WSDL Interface Description
Direct Mediator
XML Schemas
BSML Schema
AGAVE Schema
INSD-XML Schema
DDBJ-XML Schema
Semantic Web Service annotations relate WSDL message parts to concepts within an ontology
Semantic Web Service Annotation
[OWL concept]
<WSDL message>
GRIMOIRES Registry
Service's WSDL Interfaces and corresponding semantic annotations are stored in the GRIMOIRES registry
All Type Adaptors are described using WSDL and registerd with GRIMOIRES
WSDL Interface Defintion
•Types
•Messages
•Ports
•Bindings
•Services
WSDL message parts are defined using XML Schema
Web Services are described using WSDL
Web Services
XEMBL
SoapLab
DDBJ-XML
NCBI_Blast

FIGURE 4.12: High-level overview of the GRIMOIRES registry and information sources

this requirement within a bioinformatics Grid application, we found a modular transformation is difficult to achieve with complex data sets. Our solution comes in the form of FXML-M (Chapter 5): a declarative and composable mapping language with a well defined transformation process. FXML-M has novel features that allow complex mappings to be specified: predicate support is included so mapping can be specified in terms of the existence and values of other elements; mappings can be given scope so different mappings are applied depending on the context of a particular element within a document; string manipulation constructs are included through regular expressions support to allow different characters within a string value to be split and assigned to different elements.

2. **A Configurable Mediator**
   The Transformation Engine, (built with FXML-T), implements the mapping and transformation theory presented in Chapter 5 to enable the conversion of XML documents. The C-MEDIATOR (Chapter 6) combines the Transformation Engine with the ontology processing API JENA to supply a dynamically configurable Type Adaptor. The C-MEDIATOR consumes mappings that specify the processes of conceptual realisation and conceptual serialisation, along with an ontology definition in OWL, and uses them to drive the conversion of data between syntactically incongruous data formats.

3. **Architecture for the registration, sharing, and discover of Type Adaptors**
   With a mediation infrastructure in place, WS-HARMONY supports automatic workflow harmonisation through the discovery of appropriate Type Adaptors at runtime. To achieve this, all Type Adaptor components (both direct and intermediary) are described using WSDL and registered with the GRIMOIRES service registry. This means the existing GRIMOIRES query interface can be reused to support the discover of Type Adaptors according to the desired input and output types. To overcome the limitations of existing Web Service invocation APIs, such as Apache Axis and JAX-RPC, with respect to the invocation of previously unseen services, the Dynamic Web Service Invoker is used. These additional architecture components are presented in Chapter 7.

Chapter 5

Transformation Theory

To harmonise dataflow between two syntactically incompatible service interfaces, data transformations can be used to convert a data instance from one representation to another. In Chapter 4, we identified two ways in which this mediation can be performed: *direct* (where translation is performed from one format straight to another) and *intermediary-based* (where a common format is used as a *lingua franca*). Since direct mediation has poor scalability and is difficult to use in large communities where many different formats are used, we have concentrated our efforts on the intermediary-based approach because it supports better scalability and eases the introduction of new data formats (Chapter 4, Section 4.1).

By using OWL ontologies to capture the structure and semantics of data structures, OWL concept instances can be used as the intermediate representation allowing all semantically equivalent data formats to become interchangeable (Chapter 4, Section 4.1.1). Existing semantic service annotation techniques, such as OWL-S, WSDL-S and the FETA annotation model, already use the notion of a *semantic type*: a concept from an ontology which is assigned to each input and output type for a service interface. These annotations are reused within the WS-HARMONY architecture, effectively assigning each concrete type a corresponding conceptual model in OWL via the semantic type. By assuming a canonical XML representation for OWL concept instances, we simplify the transformation problem allowing realisation and serialisation transformations to be expressed as XML transformations.

The contribution of this Chapter is a modular and composable XML mapping language and translation formalisation to support the transformation of data between different formats. The novelty of the language is the combination of the following features:

- A declarative and composable mapping language to support mapping reuse and schema composition.
- Composite mappings so a single element in the source document can be mapped to a combination of elements in the destination document (and vice versa).
- Predicate support to allow elements and attributes to be mapped differently depending on their content or structure.
- Basic string manipulation so literal values can be split and mapped to different elements in a destination document.
- Scoping to allow different mappings to be applied depending on the context of an element within the document.
- XML syntax for the specification of mappings.
- A formalism to define the mapping language and transformation process, giving precise semantics for the language and the specification of an abstract implementation.

The Chapter begins in Section 5.1 where we derive our transformation requirements using data sets from our use case. In Section 5.2 we give a brief overview of XML and XML schema, showing how they are represented within an existing formalisation [26]. This formalism is then extended in Section 5.3 to describe the mechanics involved in a transformation process. Section 5.4 describes our transformation theory at a high level, before we present its formalisation in Section 5.5. In Section 5.6, example mappings for conceptual realisation of the DDBJ bioinformatics service output are presented, along with their corresponding XML syntax in Section 5.7. We conclude in Section 5.8 by summarising our transformation theory and present additional features that could be included in future work.

5.1 Transformation Requirements

We stated in Chapter 4 that we simplify the transformation requirements for conceptual serialisation and conceptual realisation by assuming a canonical XML representation of OWL concept instances. This way, we can view the translation process as an XML to XML translation. While it is common for OWL users to specify OWL concepts and their instances using XML syntax, XML schemas do not usually exist to validate them. Therefore, we automatically generate schemas using the OWL-$\mathcal{XIS}$ generator (OWL XML instance schema generator), presented in full in Chapter 7. An example instance schema for the Sequence Data ontology used in our use case can be found in Appendix C. By using this XML schema, we are able to describe an instance of the *Sequence_Data_Record* concept using the XML given in Listing 5.1.

To provide a modular transformation solution, we use mappings to express the relationship between the XML elements and attributes in a source schema, and their corresponding elements and attributes in a destination schema. We illustrate this idea in Figure 5.1 where we show a subset of a full sequence data record in DDBJ format and its corresponding OWL concept instance (serialised in XML). We consider six different mapping types, with examples given in Figure 5.1, that highlight our mapping requirements:

1. **Single element to element mapping**
   In simple cases, elements and attributes in a source schema correspond directly to elements and attributes in a destination schema. For example, in Figure 5.1, the `<DDBJXML>` element is mapped to the `<Sequence_Data_Record>` element.

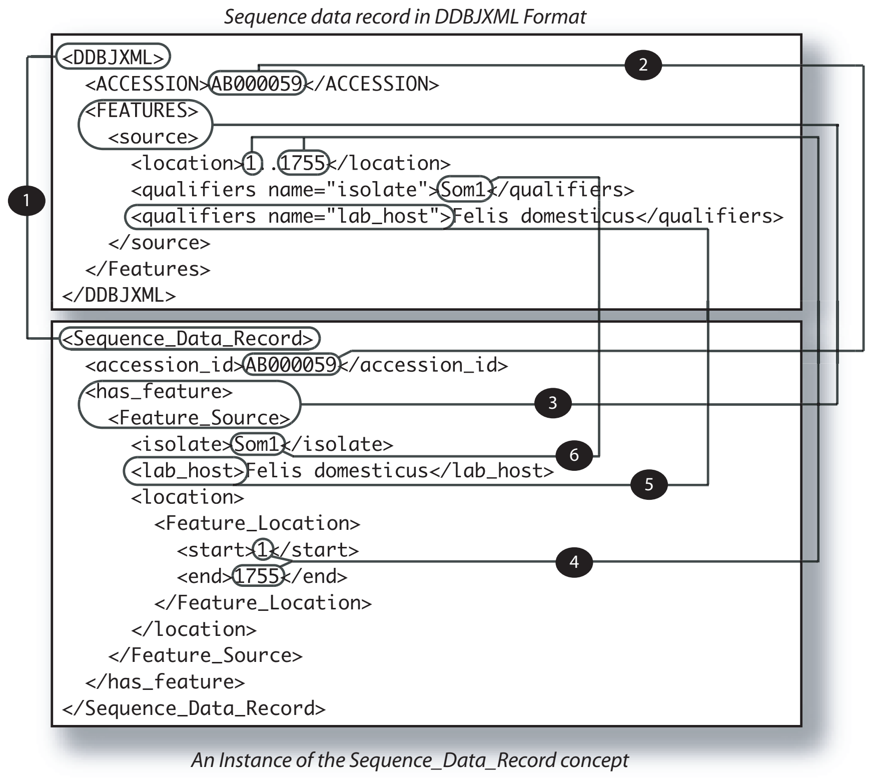
2. **Element contents mapping**
   When elements and attributes contain literal values (e.g. strings and numbers), it is necessary to copy the literal value from the source document and include it in the destination document. For example, the text value `AB000059` contained in the `<ACCESSION>` element must be copied to the destination document and inserted as the contents of the `<accession_id>` element.

```xml
<?xml version="1.0" encoding="iso−8859−1" ?>
<Sequence_Data_Record xmlns="http://www.ecs.soton.ac.uk/~mns03r/schema/Sequence−Ont">
  <accession_id>AB000059</accession_id>
  <sequence>atgagtgatggagcagttcaaccagacggtggtcaacctgctgtcagaaa...</sequence>
  <description>Feline panleukopenia virus DNA for capsid protein 2</description>
  <has_reference>
    <Reference>
      <authors>Horiuchi M.</authors>
      <journal>Submitted (22−DEC−1996) to the EMBL/GenBank/DDBJ databases.
      Motohiro Horiuchi, Obihiro University of Agriculture and Veterinary Medicine,
      Veterinary Public Health; Inada cho, Obihiro, Hokkaido 080,
      Japan (E−mail:horiuchi@obihiro.ac.jp, Tel:0155−49−5392)</journal>
    </Reference>
  </has_reference>
  <has_reference>
    <Reference>
      <authors>Horiuchi M.</authors>
      <title>evolutionary pattern of feline panleukopeina
      virus differs fromn that of canine parvovirus</title>
      <journal>Unpublished Reference</journal>
    </Reference>
  </has_reference>
  <has_feature>
    <Feature_Source>
      <location>
        <Feature_Location>
          <start>1</start>
          <end>1755</end>
        </Feature_Location>
      </location>
      <lab_host>Felis domesticus</lab_host>
      <isolate>Som1</isolate>
      <organism>Feline panleukopenia virus</organism>
    </Feature_Source>
  </has_feature>
  <has_feature>
    <Feature_CDS>
      <location>
        <Feature_Location>
          <start>1</start>
          <end>1755</end>
        </Feature_Location>
      </location>
      <translation>MSDGAVQPDGGQPAVRNERATGSGNGSGGGGGGGSGGVGISTG...</translation>
      <product>capsid protein 2</product>
    </Feature_CDS>
  </has_feature>
</Sequence_Data>
```

Listing 5.1: An XML representation for an instance of a Sequence_Data concept

3. **Multiple element mapping**

   In some cases, the relationship between elements in a source and destination schema is not atomic; a combination of elements in the source document may constitute a single element (or another combination of elements) in the destination document. For example, the `<FEATURES>` element containing a `<source>` element is mapped to the `<has_feature>` element containing a `<Feature_Source>` element in our example.


Sequence data record in DDBJXML Format
1
2
3
4
5
6
An Instance of the Sequence_Data_Record concept

```
<DDBJXML>
  <ACCESSION>AB000059</ACCESSION>
  <FEATURES>
    <source>
      <location>1..1755</location>
      <qualifiers name="isolate">Som1</qualifiers>
      <qualifiers name="lab_host">Felis domesticus</qualifiers>
    </source>
  </Features>
</DDBJXML>
```

```
<Sequence_Data_Record>
  <accession_id>AB000059</accession_id>
  <has_feature>
    <Feature_Source>
      <isolate>Som1</isolate>
      <lab_host>Felis domesticus</lab_host>
      <location>
        <Feature_Location>
          <start>1</start>
          <end>1755</end>
        </Feature_Location>
      </location>
    </Feature_Source>
  </has_feature>
</Sequence_Data_Record>
```

FIGURE 5.1: Mappings between elements and attributes in the DDBJXML Sequence Data format and elements within the XML serialisation of the `Sequence_Data_Record` OWL concept

4. **String manipulation support**

   In complex cases, the contents of a string literal may contain two or more distinct pieces for information. In Figure 5.1, the `<location>` element has text containing the start and end position, delimited by `".."`. Each of these positions must be mapped to separate elements in the destination document because they are assigned separate properties in the ontology.

5. **Predicate support**

   Sometimes, an element or attribute from a source schema may be mapped differently depending on the value of an attribute or element, or even the presence of other elements within the document. This can be seen in Figure 5.1 where the `<qualifiers>` element is mapped differently depending on the value of the `@name` attribute - in the case of Mapping 5, when the string equals `"lab_host"` , the element is mapped to the `<lab_host>` element.

6. **Local Scoping**

   In some scenarios, we may wish to map elements differently based on their context. For example, in a DDBJ record, the contents of the `<qualifiers>` element (a string value) is mapped differently depending on the value of the `@name` attribute. In mapping 6, the string contents of the `<qualifiers>` element is mapped to the contents of the `<isolate>` element. To support this kind of behaviour, our mapping language supports local scoping so different rules can be applied in different contexts.

Because of these complex mapping requirements, we specify our mapping language and the transformation of XML documents using a formalisation. This facilitates a sound and efficient implementation (presented later in Chapter 6) and helps us capture the more difficult transformation properties such as predicate support and local scoping.

5.2 XML Formalisation

We have elected to base our mapping and translation theory on an existing XML and XML schema formalisation [26] called Model Schema Language (MSL) - a W3C working draft [27]. While other XML and XML schema formalisms have been proposed [17] [79], MSL captures the most complex XML constructs such as type inheritance and cardinality constraints, as well as lending itself to the specification of mappings between different XML schemas and the process of document translation driven by such mappings.

In this Section, we outline the principal features of MSL: how elements, attributes and types are referenced (Section 5.2.1), how groups of elements are specified for type declarations (Section 5.2.2), how XML schema components[1] are defined (Section 5.2.3), and how XML documents are represented (Section 5.2.4). This will give the reader enough knowledge to understand our mapping and translation formalisation, which appears later in the Chapter in Section 5.5.

[1]We use the term components to encompass elements, attributes and literal values.

```
 1 <?xml version="1.0" encoding="iso-8859-1" ?>
 2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 3  targetNamespace="http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source"
 4  xmlns="http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source">
 5
 6   <xsd:element name="a">
 7     <xsd:complexType>
 8       <xsd:all>
 9         <xsd:element name="b" type="xsd:string" minOccurs="1" maxOccurs="1"/>
10         <xsd:element ref="c" minOccurs="1" maxOccurs="1"/>
11       </xsd:all>
12       <xsd:attribute name="id" type="xsd:string"/>
13     </xsd:complexType>
14   </xsd:element>
15
16   <xsd:element name="c" type="c-type"/>
17
18   <xsd:complexType name="c-type">
19     <xsd:sequence>
20       <xsd:element ref="b" minOccurs="1" maxOccurs="2"/>
21     </xsd:sequence>
22   </xsd:complexType>
23
24   <xsd:complexType name="c-extended">
25     <xsd:complexContent>
26       <xsd:extension base="c-type">
27         <xsd:sequence>
28           <xsd:element ref="d" minOccurs="1" maxOccurs="3">
29         </xsd:sequence>
30       </xsd:extension>
31     </xsd:complexContent>
32   </xsd:complexType>
33
34   <xsd:element name="b" type="xsd:integer"/>
35   <xsd:element name="d" type="xsd:integer"/>
36
37 </xsd:schema>
```

LISTING 5.2: A Simple XML Schema

5.2.1 Normalised schema

MSL references the components of an XML schema, such as elements, attributes and types, using a normalised format. Normalisation assigns a unique, universal name to each schema part and provides a flat representation of the components found within a schema document. This allows us to distinguish between components with the same name that have been declared within different scopes. To exemplify this notation, we provide the normalised form for all XML components declared in the simple XML schema shown in Listing 5.2 with corresponding line numbers in square brackets to show where they are defined. These references are simply used to point to XML schema components: the definition of actual elements and types is presented later in Section 5.2.2.

```
[6]  http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::a
[7]  http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::a/type::*
[12] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::a/type::*/attribute::id
[9]  http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::a/type::*/element::b
[16] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::c
[18] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#type::c-type
[24] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#type::c-extended
[34] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::b
[35] http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source/#element::d
```

The first part of the normalised schema reference, up to the first occurrence of the `#` symbol, is the namespace. The second part (following the `#` symbol) is a path of sort / name pairs (delimited by `::`), each containing a sort (e.g. `#element`, `#attribute`, or `#type`) designating the kind of component referenced, and a name (e.g. `a` or `id`) which is the local name assigned to the component. For example, the element `a` is defined in the global scope (line 6 of Listing 5.2) and is referenced with the namespace prefix `http://www.ecs.soton.ac.uk/ mns03r/schema/Example-Source` and the normalised path reference `element::a` . The element `a` contains an anonymous complex type definition (line 7) which is referenced using the path `element::a/type::*` (where "*" represents an anonymous type and should not be confused with a wild card character). This complex type has a locally defined element (line 9) named `b` which can be distinguished from the globally defined element named `b` (line 34) because they have different normalised schema references (`element::a/type::*/element::b` and `element::b` respectively). The type refinement given in line 24 is used later to illustrate type inheritance within MSL.

For compactness, a short form notation is used throughout the rest of this Chapter to refer to schema components where the namespace is dropped along with the sort definition. This allows us to reference the element `a` simply using `a`, the anonymously defined type within the scope of `a` using `a/*`, and the attribute `id` (line 12) using `a/*/@id`.

5.2.2 Model Groups

In XML, elements and attributes are assigned types to describe their contents. For elements containing data values, this is one of the pre-defined XML types such as `xsd:string` or `xsd:int`, or a *simple type* that restricts the content of an existing type (for example, numbers between 1 and 10). For elements that contain other elements, such as element `a` in our example above (Listing 5.2), their type is a *complex type*. A complex type falls into one of three categories, specified using one of the following indicators:

- `<xsd:sequence>` - contains a sequence of elements in a specified order.
- `<xsd:all>` - contains a collection of elements in any order.
- `<xsd:choice>` - contains one element from a choice of elements.

Occurrence indicators may be set to specify the number of times each content element should appear (e.g. an element in a sequence can only appear once).

In MSL, the contents of an XML type is specified by a model group using traditional regular expression notation [4]. We let g range over model groups.

group g ::=	ϵ	empty sequence
$\vert$	θ	empty choice
$\vert$	g_1, g_2	a sequence of g_1 followed by g_2
$\vert$	$g_1 \mid g_2$	choice of g_1 or g_2
$\vert$	$g_1 \& g_2$	an interleaving of g_1 and g_2 in any order
$\vert$	$g\{m, n\}$	g repeated between minimum m and maximum n times
$\vert$	$a[g]$	attribute with name a containing g
$\vert$	$e[g]$	element with name e containing g
$\vert$	p	atomic datatype (such as string or integer)
$\vert$	x	component name (in normalised form)

These model groups are used in the definition of schema components, as we describe in the following section.

5.2.3 Components

In MSL, schema components (XML elements, attributes, etc. . .) can be one of seven sorts[2]: *element*, *attribute*, *simply type*, *complex type*, *attribute group* or *model group*. We let *srt* range over sorts.

$$
\begin{array}{rcl}
\text{sort } srt & ::= & \texttt{attribute} \\
& | & \texttt{element} \\
& | & \texttt{simpleType} \\
& | & \texttt{complexType} \\
& | & \texttt{attributeGroup} \\
& | & \texttt{modelGroup}
\end{array}
$$

In XML, it is possible to express rudimentary type inheritance. When defining a type, a base type must be specified (by default this is assumed to be `xsd:UrType`). A type may either extend the base type or refine it. Extension is used in the case where the subtype allows more elements and attributes to be contained within it, such as the type `c-extended` in Listing 5.2. Refinement is used to constrict the existing elements and attributes defined by the base type, for example, by imposing more restrictive cardinality constraints.

We let *cmp* range over components where x is a reference to another normalised

[2]the term *sort* is used to avoid confusion with the XML term *type*

component name, *der* ranges over the two types of derivation (extension or refinement), *ders* is a set of *der*'s, *b* is a boolean value and *g* is a model group.

$$
\begin{array}{rcl}
\text{components } cmp & ::= & \texttt{component(} \\
 & & \quad \texttt{sort = } srt \\
 & & \quad \texttt{name = } x \\
 & & \quad \texttt{base = } x \\
 & & \quad \texttt{derivation = } der \\
 & & \quad \texttt{refinement = } ders \\
 & & \quad \texttt{abstract = } b \\
 & & \quad \texttt{content = } g \\
 & & \texttt{)}
\end{array}
$$

A *derivation* specifies how the component is derived from its base type. We let *der* range over derivations, and *ders* range over sets of derivations:

$$
\begin{array}{rcl}
\text{derivation } der & ::= & \texttt{extension} \\
 & | & \texttt{refinement} \\
\text{derivation set } ders & ::= & \{der_1, \ldots, der_l\}
\end{array}
$$

The refinement field of a component definition states the permissible derivations that can be made using this component as base. With a means to specify schema components, the components from our example schema (Listing 5.2) can be defined as in Figure 5.2 (preceded with corresponding line numbers in square brackets to indicate where they are defined in the schema listing). The content of an element or attributes is its type (e.g. element `a` has the content `a/*`), and the content of a complex type is a list of the elements and attributes it contains (e.g. type `a/*` contains an interleaving of `a/*/@id`, `a/*/b`, and c).

```
[6]
component(
 sort = element,
 name = a,
 base = xsd:UrElement,
 derivation = restriction,
 refinement = {},
 abstract = false,
 content = a/*
)

[7]
component(
 sort = complexType,
 name = a/*,
 base = xsd:UrType,
 derivation = restriction,
 refinement = {restriction,extension}
 abstract = false
 content = a/*/@id{1,1} & a/*/b{1,1} & c{1,1}
)

[12]
component(
 sort = attribute,
 name = a/*/@id,
 base = xsd:UrAttribute,
 derivation = restriction,
 refinement = {restriction}
 abstract = false
 content = xsd:string
)

[9]
component(
 sort = element,
 name = a/*/b,
 base = xsd:UrElement,
 derivation = restriction,
 refinement = {}
 abstract = false
 content = xsd:string
)

[16]
component(
 sort = element,
 name = c,
 base = xsd:UrElement,
 derivation = restriction,
 refinement = {},
 abstract = false,
 content = c-type
)

[18]
component(
 sort = complexType,
 name = c-type,
 base = xsd:UrType,
 derivation = restriction,
 refinement = {restriction, extension}
 abstract = false
 content = b{1,2}
)

[24]
component(
 sort = complexType,
 name = c-extended,
 base = c-type,
 derivation = extension,
 refinement = {restriction, extension}
 abstract = false
 content = d{1,3}
)

[34]
component(
 sort = element,
 name = b,
 base = xsd:UrElement,
 derivation = restriction,
 refinement = {},
 abstract = false,
 content = xsd:integer
)

[35]
component(
 sort = element,
 name = d,
 base = xsd:UrElement,
 derivation = restriction,
 refinement = {},
 abstract = false,
 content = xsd:integer
)
```

Figure 5.2: msl to represent the schema components defined in Listing 5.2 with listing line numbers for components indicated in square brackets.

5.2.4 Typed Documents

In the previous Sections (5.2.1, 5.2.2 and 5.2.3), we have described how MSL can be used to specify XML schema components. To represent instances of the schema components, or XML documents, MSL uses *typed documents*. We let td range over typed documents:

document td ::=	ϵ	empty document
$\mid$	td_1, td_2	a sequence of typed documents
$\mid$	c	a constant (e.g. a string or an integer)
$\mid$	$a[s \ni c]$	an attribute a of type s with contents c
$\mid$	$e[t \ni td]$	an element e of type t with contents td

As an example, Figure 5.3 contains MSL to express the XML document given in Listing 5.3 adhering to the schema presented earlier in Listing 5.2. The root element `a`, of type `a/*`, is a sequence containing the attribute `a/*/@id` (with the string value `"foo"`), the element `a/*/b` (with the string value `"bar"`), and the element `c`. The element `c`, of type `c-type`, contains a sequence with two `b` elements each containing the integer values `1` and `2`.

```
a[a/* ∋
  a/*/@id[xsd:string ∋ "foo"],
  a/*/b[xsd:string ∋ "bar"],
  c[c-type ∋
    b[xsd:integer ∋ 1],
    b[xsd:integer ∋ 2]
  ]
]
```

FIGURE 5.3: MSL to express the XML document given in Listing 5.3

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<a xmlns="http://www.ecs.soton.ac.uk/~mns03r/schema/Example-Source" id="foo">
  <b>bar</b>
  <c>
    <b>1</b>
    <b>2</b>
  </c>
</a>
```

LISTING 5.3: An example xml document

5.3 Formalisation Extensions

Before describing our XML mapping and transformation methodology, we present two extensions to the MSL formalisation: we describe a notion of *document paths*, which allow us to specify a selection of components from within an XML document, and simple predicates which will be used later to specify conditional mappings.

5.3.1 Document Paths

To specify a selection of child elements, attribute or literal values located deep within a given typed document, we use a *document path*. This is an important XML construct and is already implemented in XPATH [35]. However, XPATH has not been formalised within MSL, so we present our own simple document path formalism. We let path components θ range over attribute names, element names, the keyword *value*, the keyword *value* with a regular expression, and the empty document ϵ:

$$\begin{array}{rll}
\text{path component } \theta ::= & a & \text{attribute name} \\
| & e & \text{element name} \\
| & value & \text{value extraction} \\
| & value\{regexp\} & \text{regular expression} \\
| & \epsilon & \text{empty document} \\
\text{regular expression } regexp ::= & string &
\end{array}$$

The empty document ϵ is included so empty XML elements (e.g. `<x/>` can be matched). A path expression is then specified by a sequence of path components.

$$\text{path expression } \Theta \;=\; \langle \theta_1, \theta_2, \ldots, \theta_n \rangle$$

Definition 1 (Path Components) *To evaluate a path expression Θ against a source typed document td_s, each path component (θ_n) in the expression must match components within td_s. Given a typed document td_s that contains the components*

td_m that match θ, we write:

$$\theta \vdash td_s \rightarrow td_m$$

To define this behaviour, and others throughout the rest of this Chapter, we use inference rule notation [47]. In this notation, when all statements above the line hold, then the statement below the line also holds. We present rules to define the matching of path components against typed documents in Figure 5.4. Rule `PATHC.A` states that a path component θ referencing an attribute a matches the typed document $\mathrm{a}[\mathrm{t} \ni td_c]$, and therefore $\theta \vdash td_s \rightarrow \mathrm{a}[\mathrm{t} \ni td_c]$ holds. Rule `PATHC.E` uses the same principle to define the matching of elements. `PATHC.C` states that a path component $\theta = value$ will match a typed document only if it is a constant value (i.e. $td_s = c$). To match regular expressions against constants (rule `PATHC.REG`), we assume the existence of a function $\mathbf{eval}(regexp, c) = r$ which evaluates the regular expression $regexp$ against the string c giving the result r. The matching of the empty document is defined in rule `PATHC.EMP`. Rules `NOT.PATHC.A`, `NOT.PATHC.E`, `NOT.PATHC.C`, `NOT.PATHC.REG`, and `NOT.PATHC.EMP` define the cases where the path component θ is not matched against the typed document td_s, so $\theta \vdash td_s \rightarrow \perp$ holds. When matching any path component against a typed document that is a sequence of other typed documents, there are four possible cases: only the first element in the sequence is matched (`PATHC.SA`), only the second element in the sequence is matched (`PATHC.SB`), both elements are matched (`PATHC.SAB`), or neither element is matched (`NOT.PATHC.S`).

Definition 2 (Child Documents) *When evaluating a path expression, each path component is matched in order against components in the source document. To traverse into the document and take direct children of an element or attribute, a notion of typed document contents is required. The direct child of a parent typed document td_p is a child typed document td_c and is denoted by:*

$$\mathbf{child}(td_p) = td_c$$

To evaluate a path expression (which is a sequence of path components), it is necessary to take the contents of an element or attribute so it can be evaluated against the next path component in the sequence. Inference rules to describe this behaviour are given in Figure 5.5. Rule `CHILD.A` states that a typed document

$$\texttt{PATHC.A} \quad \frac{\theta = \mathrm{a} \quad td_s = \mathrm{a}[\mathrm{t} \ni td_c]}{\theta \vdash td_s \rightarrow \mathrm{a}[\mathrm{t} \ni td_c]}$$

$$\texttt{NOT.PATHC.A} \quad \frac{\theta = \mathrm{a} \quad td_s \neq \mathrm{a}[\mathrm{t} \ni td_c]}{\theta \vdash td_s \rightarrow \perp}$$

$$\texttt{PATHC.E} \quad \frac{\theta = \mathrm{e} \quad td_s = \mathrm{e}[\mathrm{t} \ni td_c]}{\theta \vdash td_s \rightarrow \mathrm{e}[\mathrm{t} \ni td_c]}$$

$$\texttt{NOT.PATHC.E} \quad \frac{\theta = \mathrm{e} \quad td_s \neq \mathrm{e}[\mathrm{t} \ni td_c]}{\theta \vdash td_s \rightarrow \perp}$$

$$\texttt{PATHC.C} \quad \frac{\theta = value \quad td_s = c}{\theta \vdash td_s \rightarrow c}$$

$$\texttt{NOT.PATHC.C} \quad \frac{\theta = value \quad td_s \neq c}{\theta \vdash td_s \rightarrow \perp}$$

$$\texttt{PATHC.REG} \quad \frac{\theta = value\{regexp\} \quad td_s = c \quad \mathbf{eval}(regexp, c) = r}{\theta \vdash td_s \rightarrow r}$$

$$\texttt{NOT.PATHC.REG} \quad \frac{\theta = value\{regexp\} \quad td_s \neq c}{\theta \vdash td_s \rightarrow \perp}$$

$$\texttt{PATHC.EMP} \quad \frac{\theta = \epsilon \quad td_s = \epsilon}{\theta \vdash td_s \rightarrow \epsilon}$$

$$\texttt{NOT.PATHC.EMP} \quad \frac{\theta = \epsilon \quad td_s \neq \epsilon}{\theta \vdash td_s \rightarrow \perp}$$

$$\texttt{PATHC.SA} \quad \frac{\theta \quad td_s = td_a, td_b \quad \theta \vdash td_a \rightarrow td_r \quad \theta \vdash td_b \rightarrow \perp}{\theta \vdash td_s \rightarrow td_r}$$

$$\texttt{PATHC.SB} \quad \frac{\theta \quad td_s = td_a, td_b \quad \theta \vdash td_a \rightarrow \perp \quad \theta \vdash td_b \rightarrow td_r}{\theta \vdash td_s \rightarrow td_r}$$

$$\texttt{PATHC.SAB} \quad \frac{\theta \quad td_s = td_a, td_b \quad \theta \vdash td_a \rightarrow td_p \quad \theta \vdash td_b \rightarrow td_q}{\theta \vdash td_s \rightarrow td_p, td_q}$$

FIGURE 5.4: Rules to define the application of path components to typed documents

$$\texttt{CHILD.A} \quad \frac{td_s = \mathrm{a}[\mathrm{t} \ni td_c]}{\mathbf{child}(\mathrm{a}[\mathrm{t} \ni td_c]) = td_c}$$

$$\texttt{CHILD.E} \quad \frac{td_s = \mathrm{e}[\mathrm{t} \ni td_c]}{\mathbf{child}(\mathrm{e}[\mathrm{t} \ni td_c]) = td_c}$$

$$\texttt{CHILD.C} \quad \frac{td_s = \mathrm{c}}{\mathbf{child}(\mathrm{c}) = \mathrm{c}}$$

$$\texttt{CHILD.EMP} \quad \frac{td_s = \epsilon}{\mathbf{child}(\epsilon) = \epsilon}$$

$$\texttt{CHILD.SEQ} \quad \frac{td_s = td_a, td_b}{\mathbf{child}(td_a, td_b) = td_a, td_b}$$

FIGURE 5.5: Rules to define the direct children of typed documents

td_s that is the attribute definition a[t $\ni td_c$] contains the document td_c. A similar definition is used to define the contents of an element in rule `CHILD.E`. The other three rules define the contents of the empty document (`CHILD.EMP`), a constant (`CHILD.C`), and a sequence of typed documents (`CHILD.SEQ`) to be itself.

Definition 3 (Path Expressions) *The application of path expression Θ to a typed document td_s yields a result typed document td_r. This action represents the extraction of elements deep within a typed document according to the path components specified in the path expression. To denote this, we write:*

$$\Theta \vdash td_s \rightarrow td_r$$

With rules in place to describe the contents of typed documents and the matching of path components, the evaluation of a path expression can be specified as in Figure 5.6. The result document, td_n, is taken from the contents of the final component matched ($\mathbf{child}(td_{n-1'}) = td_n$).

As an example, the path expression $\Theta = \langle \texttt{a}, \texttt{a}/*/@\texttt{id}, value \rangle$ can be evaluated against the typed document given in Figure 5.3 to give the result `"foo"`, and would be equivalent to applying the XPATH statement `a/@id/text()`. To illustrate this

$$\texttt{PATH.EVAL}\quad \frac{\begin{array}{c}\Theta = \langle \theta_1, \theta_2, \ldots, \theta_n \rangle \\ \theta_1 \vdash td_s \rightarrow td_{s'} \quad \mathbf{child}(td_{s'}) = td_1, \\ \theta_2 \vdash td_1 \rightarrow td_{1'} \quad \mathbf{child}(td_{1'}) = td_2, \\ \ldots, \\ \theta_n \vdash td_{n-1} \rightarrow td_{n-1'} \quad \mathbf{child}(td_{n-1'}) = td_n \end{array}}{\Theta \vdash td_1 \rightarrow td_n}$$

FIGURE 5.6: A rule to define the application of a path expression to a typed document

evaluation, Figure 5.7 shows the steps involved with and explanation of the rules used below:

1. The source document is td_s and the path expression is Θ. Rather than write the full typed document, ... is used to denote element and attribute contents. Rule `PATH.EVAL` is used to derive the result document and is comprised of three steps: α, β, and γ, each denoting the application of a path component from Θ (e.g. $[\alpha]$) and its child document (e.g. $[\alpha']$).

2. $[\alpha]$ - The first path component in Θ is matched against the root document ($\texttt{a} \vdash td_s \rightarrow \texttt{a}[\texttt{a}/* \ni \ldots]$) using rule `PATHC.E`.

3. $[\alpha']$ - The direct child of the matched document is found using rule `CHILD.E`. The direct child is a sequence of typed documents containing the attribute `a/*/@id`, the element `a/*/b`, and the element `c`.

4. $[\beta]$ - The second path component in Θ is then matched against the sequence using rule `PATHC.SA` since only the first document in the sequence matches (rule `PATHC.A`) and the remaining two do not (rules `NOT.PATH.A` and `NOT.PATH.S`).

5. $[\beta']$ - The direct child of the matched document is found using rule `CHILD.A`. The direct child of the attribute is the literal value `foo`.

6. $[\gamma]$ - The final path component in Θ is matched against the literal value using rule `PATHC.C` ($value \vdash "foo" \rightarrow "foo"$).

7. $[\gamma']$ - The direct child of the literal value is itself (from rule `CHILD.C`) and is the final result of the application of the path expression Θ to td_s.

$$(\mathtt{PATH.EVAL}) \quad \frac{\begin{array}{c} td_s = \mathtt{a}[\mathtt{a}/* \ni \ldots] \\ \Theta = \langle \mathtt{a}, \mathtt{a}/*/@\mathtt{id}, value \rangle \\ [\alpha] \quad [\alpha'] \\ [\beta] \quad [\beta'] \\ [\gamma] \quad [\gamma'] \end{array}}{\Theta \vdash td_s \rightarrow \mathtt{foo}}$$

$$(\mathtt{PATHC.E}) \quad [\alpha] \quad \frac{\theta = \mathtt{a} \quad td_s = \mathtt{a}[\mathtt{a}/* \ni \ldots]}{\mathtt{a} \vdash td_s = \mathtt{a}[\mathtt{a}/* \ni \ldots]}$$

$$(\mathtt{CHILD.E}) \quad [\alpha'] \quad \frac{td_s = \mathtt{a}[\mathtt{a}/* \ni \ldots]}{\mathbf{child}(\mathtt{a}[\mathtt{a}/* \ni \ldots]) = \mathtt{a}/*/@\mathtt{id}[\ldots], \mathtt{a}/*/\mathtt{b}[\ldots], \mathtt{c}[\ldots]}$$

$$(\mathtt{PATHC.SA}) \quad [\beta] \quad \frac{\dfrac{\theta = \mathtt{a}/*/@\mathtt{id} \quad td_s = \mathtt{a}/*/@\mathtt{id}[\ldots]}{\mathtt{a}/*/@\mathtt{id} \vdash \mathtt{a}/*/@\mathtt{id}[\ldots] \rightarrow \mathtt{a}/*/@\mathtt{id}[\ldots]} \quad \dfrac{\dfrac{\theta = \mathtt{a}/*/@\mathtt{id} \quad td_s = \mathtt{a}/*/\mathtt{b}[\ldots]}{\mathtt{a}/*/@\mathtt{id} \vdash \mathtt{a}/*/\mathtt{b}[\ldots] \rightarrow \perp} \quad \dfrac{\theta = \mathtt{a}/*/@\mathtt{id} \quad td_s = \mathtt{c}[\ldots]}{\mathtt{a}/*/@\mathtt{id} \vdash \mathtt{c}[\ldots] \rightarrow \perp}}{\mathtt{a}/*/@\mathtt{id} \vdash \perp, \perp \rightarrow \perp}}{\mathtt{a}/*/@\mathtt{id} \vdash \mathtt{a}/*/@\mathtt{id}[\ldots], \perp \rightarrow \mathtt{a}/*/@\mathtt{id}[\ldots]}$$

$$(\mathtt{CHILD.A}) \quad [\beta'] \quad \frac{td_s = \mathtt{a}/*/@\mathtt{id}[\ldots]}{\mathbf{child}(\mathtt{a}/*/@\mathtt{id}[\mathtt{xsd:string} \ni \mathtt{foo}]) = \mathtt{foo}}$$

$$(\mathtt{PATHC.C}) \quad [\gamma] \quad \frac{\theta = value \quad td_s = \mathtt{foo}}{value \vdash \mathtt{foo} \rightarrow \mathtt{foo}}$$

$$(\mathtt{CHILD.C}) \quad [\gamma'] \quad \frac{td_s = \mathtt{foo}}{\mathbf{child}(\mathtt{foo}) = \mathtt{foo}}$$

FIGURE 5.7: An example path expression evaluation to retrieve the contents of an attribute

5.3.2 Simple Predicates

To cope with complex mappings where the semantics of an element or attribute vary depending on the existence of other elements or their values, predicate support is necessary. This notion was presented earlier in Section 5.1, example mapping 5, where the `<qualifiers>` element is mapped differently depending on the value of the `@name` attribute. We let predicate atoms *patom* range over path expressions and constants (such as a string or a number):

predicate atom *patom*	::=	Θ	path expressions
	|	c	constant

A predicate ψ is then defined as:

predicate ψ	::=	$\exists\, patom$	Evaluation of *patom* is not the empty document ϵ
	|	ψ_1 && ψ_2	Evaluation of both ψ_1 *and* ψ_2 must be true
	|	$\psi_1 \mid\mid \psi_2$	Evaluation of either ψ_1 *or* ψ_2 must be true
	|	$patom_1 \; < \; patom_2$	The evaluation of $patom_1$ is less than the evaluation of $patom_2$
	|	$patom_1 \; > \; patom_2$	The evaluation of $patom_1$ is greater than the evaluation of $patom_2$
	|	$patom_1 \; = \; patom_2$	The evaluation of $patom_1$ is equal to the evalaluation of $patom_2$
	|	$\neg\, \psi'$	The evaluation of ψ is false
	|	`true`	Always true

Definition 4 (Predicate Evaluation) *Predicates can be used to: check for existence of elements and attributes located within a typed document; the comparison of literal values against each other; and the comparison of literal values to defined constants. A predicate atom (patom) can be applied to a typed document td_s to give a result document td_r and is written:*

$$\mathbf{apply}(patom, td_s) = td_r$$

$$\texttt{PEXPR.TD} \quad \frac{\Theta \vdash td_s \rightarrow td_r}{\mathbf{apply}(\Theta, td_s) = td_r}$$

$$\texttt{PEXPR.C} \quad \frac{td_s}{\mathbf{apply}(c, td_s) = c}$$

Figure 5.8: Rules to define the evaluation of predicate expressions.

The evaluation of a predicate ψ against a typed document td_s is either true or false:

$$\psi \vdash td_s \rightarrow b$$

Since predicate atoms range over path expressions and constants, we specify two rules (`PEXPR.TD` and `PEXPR.C` in Figure 5.8) to define their evaluation against a typed document. Rule `PEXPR.TD` states that when a predicate atom *patom* is equal to a path expression Θ, and $\Theta \vdash td_s \rightarrow td_r$ (from rule `PATH.EVAL`), then the evaluation of *patom* against td_s is equal to td_r. When a predicate atom *patom* is equal to a constant c, the evaluation of *patom* to c is the constant itself (rule `PEXPR.C`). This rule is used when a comparison is made to a defined constant, e.g. the value of an element must be greater than 10.

Rules to define the evaluation of predicates are given in Figure 5.9. Rule `PEVAL.E` states that the evaluation of the predicate atom *patom* against td_s must not equal the empty document. This predicate can be used to check for the existence of elements and attributes. Rule `PEVAL.NEG` states that the evaluation of the predicate ψ' against td_s must be false. Rule `PEVAL.AND` states that the evaluation of both predicates ψ_1 and ψ_2 must be true. Rule `PEVAL.OR` states that the evaluation of either predicate ψ_1 or ψ_2 must be true. Rule `PEVAL.LESS` states that the evaluation of $patom_a$ to td_s must be less than the evaluation of $patom_b$ to td_s. Rule `PEVAL.GR` states that the evaluation of $patom_a$ to td_s must be more than the evaluation of $patom_b$ to td_s. Rule `PEVAL.EQ` states that the evaluation of $patom_a$ to td_s must be equal to the evaluation of $patom_b$ to td_s.

$$\texttt{PEVAL.E} \quad \frac{\psi = \exists\, patom \quad \mathbf{apply}(patom, td_s) = td_r \quad td_r \neq \perp}{\psi \vdash td_s \rightarrow true}$$

$$\texttt{PEVAL.NEG} \quad \frac{\psi = \neg\, \psi' \quad \psi' \vdash td_s \rightarrow false}{\psi \vdash td_s \rightarrow true}$$

$$\texttt{PEVAL.AND} \quad \frac{\psi = \psi_a \mathbin{\&\&} \psi_b \quad td \quad \psi_a \vdash td_s \rightarrow b_a \quad \psi_b \vdash td_s \rightarrow b_b}{\psi \vdash td_s \rightarrow b_a \wedge b_b}$$

$$\texttt{PEVAL.OR} \quad \frac{\psi = \psi_a \mathbin{||} \psi_b \quad td \quad \psi_a \vdash td_s \rightarrow b_a \quad \psi_b \vdash td_s \rightarrow b_b}{\psi \vdash td_s \rightarrow b_a \vee b_b}$$

$$\texttt{PEVAL.LESS} \quad \frac{\begin{array}{c}\psi = patom_a < patom_b \\ \mathbf{apply}(patom_a, td_s) = c_a \\ \mathbf{apply}(patom_b, td_s) = c_b\end{array}}{\psi \vdash td_s \rightarrow c_a < c_b}$$

$$\texttt{PEVAL.GR} \quad \frac{\begin{array}{c}\psi = patom_a > patom_b \\ \mathbf{apply}(patom_a, td_s) = c_a \\ \mathbf{apply}(patom_b, td_s) = c_b\end{array}}{\psi \vdash td_s \rightarrow c_a > c_b}$$

$$\texttt{PEVAL.EQ} \quad \frac{\begin{array}{c}\psi = patom_a = patom_b \\ \mathbf{apply}(patom_a, td_s) = c_a \\ \mathbf{apply}(patom_b, td_s) = c_b\end{array}}{\psi \vdash td \rightarrow c_a = c_b}$$

FIGURE 5.9: Rules to define the evaluation of predicates.

5.4 Transformation Process

When using the MSL formalisation of XML, we view the transformation process as an action which consumes a source document, td_s and produces a destination document, td_d. Since typed documents are specified in a hierarchical manner, with element and attribute documents containing other typed documents, we can view an XML document as a tree structure with nodes corresponding to XML components, and edges corresponding to XML types. This is illustrated in Figure 5.10 where three representations of the same XML document are given, one in standard

Example Typed Document

```
a [ a/* ∋
    b [ xsd:string ∋ "val1" ] ,
    b [ xsd:string ∋ "val2" ]
]
```

Typed Document in atomic form

td_1 = a [a/* ∋ td_2]
td_2 = td_3 , td_4
td_3 = b [xsd:string ∋ td_5]
td_4 = b [xsd:string ∋ td_6]
td_5 = "val1"
td_6 = "val2"

Typed Document in tree form

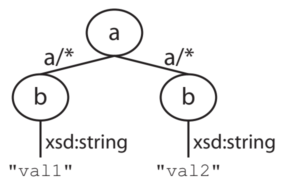
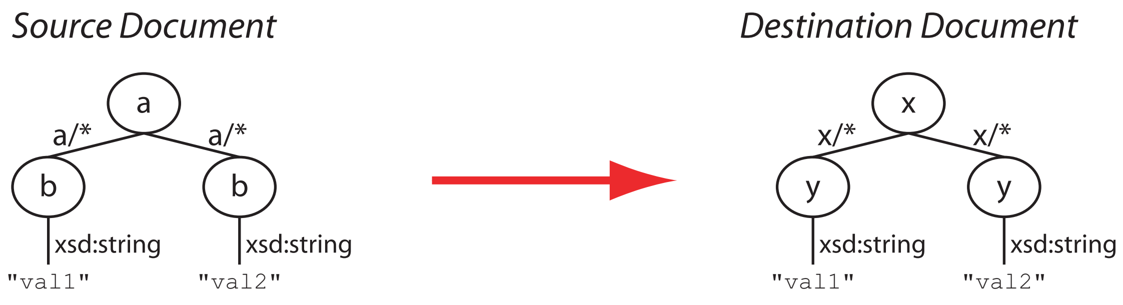


FIGURE 5.10: Viewing a typed document as a tree

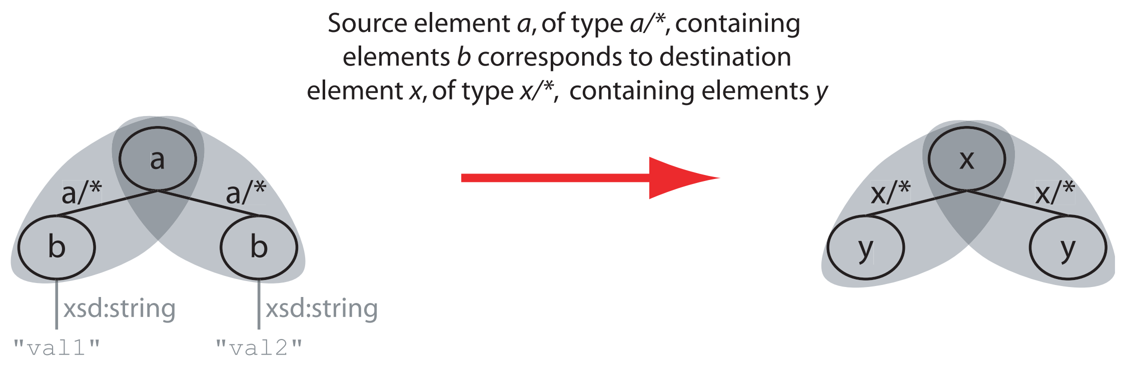
typed document notation, one with each typed document specified individually, and finally a tree representation. By viewing an XML document as a tree, we can visualise the transformation process using a recursion over the source document where groups of elements, attributes or constant values correspond directly to groups of elements, attributes or constant values in the destination document. This idea is presented visually in Figure 5.11 using a trivial transformation. With this method of transformation, we can describe a translation using a number of *mappings* which relate components in the source schema to components in the destination schema. At each stage of the recursion over the source document, mappings are used to create the appropriate parts in the destination document. We define this process formally in section 5.5 where we also describe more complex mapping constructs.

5.5 Mappings and the Transformation Process

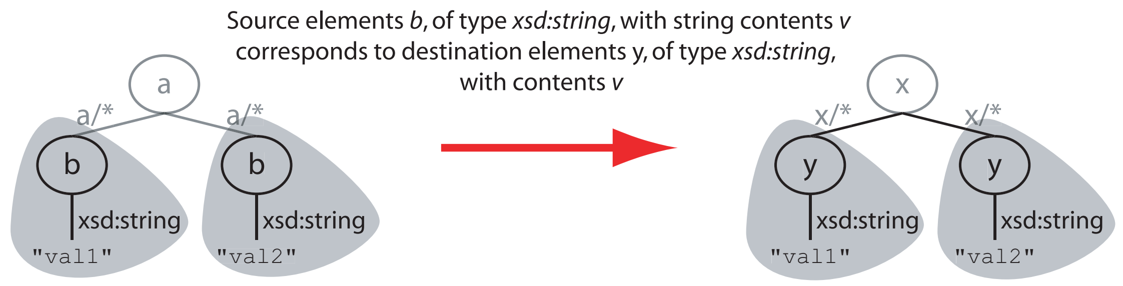
In this Section, we describe the specification of mappings and how mappings are used to direct a transformation. First, we define two kinds of mapping path: *source mapping paths* and *destination mapping paths*. Source mapping paths are used to specify the selection of components from the source document and destination mapping paths are used to describe the creation of components in the destination document.



(a) Desired Transformation


Source element *a*, of type *a/**, containing elements *b* corresponds to destination element *x*, of type *x/**, containing elements *y*
a
a/*
a/*
b
b
xsd:string
xsd:string
"val1"
"val2"
x
x/*
x/*
y
y

(b) Translation Step 1


Source elements *b*, of type *xsd:string*, with string contents *v* corresponds to destination elements y, of type *xsd:string*, with contents *v*
a
a/*
a/*
b
b
xsd:string
xsd:string
"val1"
"val2"
x
x/*
x/*
y
y
xsd:string
xsd:string
"val1"
"val2"

(c) Translation Step 2

FIGURE 5.11: Transformation through recursion

5.5.1 Mapping Paths

A source mapping path ρ is defined as a sequence of source mapping pairs:

$$\rho = \langle [\theta_1 \times \psi_1],\ [\theta_2 \times \psi_2],\ \ldots\ , [\theta_n \times \psi_n] \rangle$$

θ ranges over path components

ψ ranges over predicates

Definition 5 (Source Mapping Pairs) *Each pair in a source mapping path contains a path component (θ) that matches* XML *components from the source*

$$\texttt{SMPAIR}\quad \frac{[\theta \times \psi] \quad \theta \vdash td_s \rightarrow td_m \quad \psi \vdash td_m \rightarrow true}{[\theta \times \psi] \vdash td_s \rightarrow td_m}$$

$$\texttt{SMPATH}\frac{\begin{array}{l}\rho = \langle [\theta_1 \times \psi_1],\ [\theta_2 \times \psi_2],\ \ldots\ ,\ [\theta_n \times \psi_n] \rangle \\ [\theta_1 \times \psi_1] \vdash td_s \rightarrow td_{s'} \quad \mathbf{child}(td_{s'}) = td_1, \\ [\theta_2 \times \psi_2] \vdash td_1 \rightarrow td_{1'} \quad \mathbf{child}(td_{1'}) = td_2, \\ \qquad \ldots\ , \\ [\theta_n \times \psi_n] \vdash td_{n-1} \rightarrow td_{n-1'} \quad \mathbf{child}(td_{n-1'}) = td_n \end{array}}{\rho \vdash td_s \rightarrow td_n}$$

FIGURE 5.12: Rules to define the evaluation of source mapping paths.

document, and a predicate (ψ) that must evaluate to true. This pairing technique allows any part of a source mapping path to be assigned a predicate so complex component selections can be made. The evaluation of a source mapping pair $[\theta \times \psi]$ against a typed document td_s results in a matched document td_m and is written:

$$[\theta \times \psi] \vdash td_s \rightarrow td_m$$

Definition 6 (Source Mapping Paths) *The evaluation of a source mapping path ρ against a source document td_s yields a result document td_r (the components successfully selected by ρ) and is written:*

$$\rho \vdash td_s \rightarrow td_r$$

Figure 5.12 contains the two rules that define source mapping path evaluation. Rule `SMPAIR` states that when the path component θ matches td_s with td_m and the predicate ψ applied to those matched components evaluates to true, then $[\theta \times \psi] \vdash td_s \rightarrow td_m$ holds. The application of source mapping path (or a sequence of source mapping path pairs) can then be describe by the rule `SMPATH`.

When defining the creation of components in the destination document a joining operator is used. We let ω range over joining operators:

$$\begin{array}{lcr} \text{joining operator } \omega ::= & & \textit{join} \\ & | & \textit{branch} \end{array}$$

A destination mapping path, δ, is used to specify the creation of elements, attributes and values in the destination document, and is defined as a sequence of destination mapping pairs:

$$\delta \;=\; \langle [\theta_1 \times \omega_1],\ [\theta_2 \times \omega_2],\ \ldots\ ,\ [\theta_n \times \omega_n] \rangle$$

Each pair contains a path expression θ_n which describes the elements, attributes and values to be created, and a joining operator ω_n. The evaluation of destination mapping paths is done during the transformation process and is described in Section 5.5.2, as is the joining operator.

5.5.2 Mappings and Bindings

A *mapping* describes a selection of nodes from a source document and their corresponding representation in a destination document. We let m range over mappings:

$$\text{mapping } m \;::=\; \langle \rho,\ \delta,\ B \rangle$$

ρ is the source mapping path, δ is the destination mapping path, and B is a local binding containing mappings that should only be considered for application when the parent mapping has been applied. A *binding*, B, is defined as a sequence of mappings:

$$\text{binding } B \;::=\; \langle m_1, m_2, \ldots, m_n \rangle$$

A binding can be constructed from any number of mappings to describe the translation of components within a source document to components in a destination document. A binding is defined using a sequence because the order in which the mappings are defined is the order in which they are applied.

5.5.3 Transformation

The application of a Binding to a typed document gives the destination typed document which is the result of all compatible mapping applications. This transformation process is split into four stages:

1. **Mapping selection**
   Given td_s and a binding B, identify mappings from B that are compatible for application to td_s.

2. **Source Document Selection**
   Given the set of applicable mappings M_a, and a source document td_s, for each mapping $m_x \in M_a$ the source mapping path ρ from m_x is applied to give a result document $\rho \vdash td_s \rightarrow td_r$.

3. **Recursion**
   The result of each source mapping path (td_r) is itself translated using B to give $td_{r'}$ (where local mappings defined in the parent mapping are added to the global binding B and their ordering is preserved). The recursion continues until no mappings are valid, the empty document is encountered, or a constant value is found.

4. **Destination Document Construction**
   For each mapping applied, the destination mapping path δ is evaluated and used to create new components in the destination document. The contents of each new component created is the result of the recursive call.

Definition 7 (Mapping Compatibility) *When a mapping m can be applied to a typed document td, we write:*

$$\textbf{isCompatible}(m, td)$$

The rule `COMP.ME` in Figure 5.13 states that when the first component referenced in a source mapping path is the element e, and the source document td is the element e, then mapping m can be applied to td. Rule `COMP.MA` is similarly defined for attribute compatibility. As in the MSL formalism, we assume the existence of a fixed

$$\texttt{COMP.ME}\quad \frac{m = \langle \rho,\ \delta,\ B_l \rangle \quad \rho = \langle [e \times \psi],\ \ldots \rangle \quad td = e[t \ni td_c]}{\mathbf{isCompatible}(m, td)}$$

$$\texttt{COMP.MA}\quad \frac{m = \langle \rho,\ \delta,\ B_l \rangle \quad \rho = \langle [a \times \psi],\ \ldots \rangle \quad td = a[t \ni td_c]}{\mathbf{isCompatible}(m, td)}$$

FIGURE 5.13: Rules to define mapping compatibility

dereferencing map that takes a component name x and gives the corresponding component, so features of the component (such as its type) can be determined:

$$\begin{aligned} \mathbf{deref}(x) &= cmp \\ \text{e.g. } \mathbf{deref}(x).type &= t \\ \text{e.g. } \mathbf{deref}(x).sort &= \texttt{element} \end{aligned}$$

The most complex stage in the translation process is to construct the destination typed document. This stage is complicated because we have to handle the creation of multiple elements in order to map components from the source domain to multiple components in the destination schema. We illustrate this problem in Figure 5.14 where we show the translation of a simple source document to two possible destination documents. The destination documents differ only by the joining of element y. In the left translation, the destination mapping path $\langle [x \times join],\ [y \times join],\ [z \times branch] \rangle$ indicates that all elements discovered by the application of $\langle [a \times true],\ [b \times true] \rangle$ (or elements a which contain elements b) should be translated to elements z contained within a single element y, contained within the element x. The right translation shows a similar mapping but with unique y elements created for each match.

Definition 8 (Destination Creation Pairs) *During the transformation process, source mapping paths (ρ) are applied to the source document (td_s) to select* XML *components (written $\rho \vdash td_s \longrightarrow td_c$ from rule* ***SMPATH****). The result typed document (td_c), is paired with the destination mapping path (δ) to give a destination creation pair $P = [\delta \times td_c]$ where δ are the components to construct and td_c is their content.*

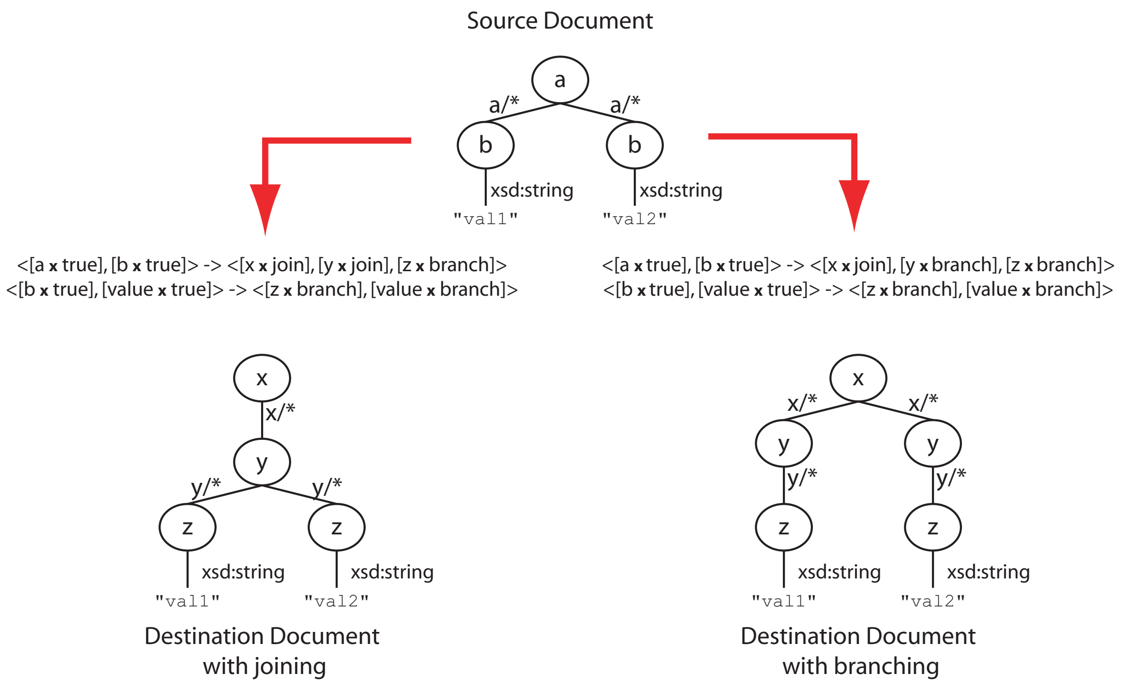

Source Document
a
a/*
a/*
b
b
xsd:string
xsd:string
"val1"
"val2"
<[a x true], [b x true]> -> <[x x join], [y x join], [z x branch]>
<[b x true], [value x true]> -> <[z x branch], [value x branch]>
<[a x true], [b x true]> -> <[x x join], [y x branch], [z x branch]>
<[b x true], [value x true]> -> <[z x branch], [value x branch]>
x
x/*
y
y/*
y/*
z
z
xsd:string
xsd:string
"val1"
"val2"
Destination Document with joining
x
x/*
x/*
y
y
y/*
y/*
z
z
xsd:string
xsd:string
"val1"
"val2"
Destination Document with branching

FIGURE 5.14: A Source Document with two possible transformations, each using a different joining operator

To denote the construction of the destination document, we write:

$$\textbf{construct}([\delta \times td_c]) = td_r$$

For the base case, when the destination mapping path δ in P contains only one destination mapping pair ($\delta = \langle[\theta{\times}\omega]\rangle$), P can construct the destination document by the rules shown in Figure 5.15. Rule `BPAIR.EVAL.E` states that When $P = [\delta \times td_c]$ and $\delta = \langle[e \times branch]\rangle$, the destination document contains the element e, of type t, with the contents td_c. Rules `BPAIR.EVAL.A`, `BPAIR.EVAL.C`, and `BPAIR.EVAL.EMP` define the construction of attributes, constants, and the empty document in a similar way.

Definition 9 (Destination Creation Set) *During the transformation process, multiple mappings may be applied to a given source document. Each mapping is applied independently to give a destination creation pair (P) that are combined to form a destination creation set $R = \{P_1, P_2, \ \dots \ , P_n\}$. When creating elements in the destination document, joining operators define whether a set of the same elements should be combined to form one element (join) or used to create a sequence of elements (branch). Therefore, a destination creation set R can be split*

$$\texttt{BPAIR.EVAL.E} \quad \frac{P = [\delta \times td_c] \quad \delta = \langle [e \times branch] \rangle \quad \mathbf{deref}(e)\texttt{.type} = t}{\mathbf{construct}(P) = e[t \ni td_c]}$$

$$\texttt{BPAIR.EVAL.A} \quad \frac{P = [\delta \times td_c] \quad \delta = \langle [a \times branch] \rangle \quad \mathbf{deref}(a)\texttt{.type} = t}{\mathbf{construct}(P) = a[t \ni td_c]}$$

$$\texttt{BPAIR.EVAL.C} \quad \frac{P = [\delta \times c] \quad \delta = \langle [value \times branch] \rangle}{\mathbf{construct}(P) = c}$$

$$\texttt{BPAIR.EVAL.EMP} \quad \frac{P = [\delta \times \epsilon] \quad \delta = \langle [\epsilon \times branch] \rangle}{\mathbf{construct}(P) = \epsilon}$$

FIGURE 5.15: Rules to define the construction of destination documents (base case).

into two subsets: R_{join} (where all destination creation pairs P have the joining operator join in the first destination mapping pair), and R_{branch} (where all destination creation pairs P have the joining operator branch in the first destination mapping pair). To denote this, we write:

$$R = R_{join} \cup R_{branch}$$

Figure 5.16 contains rules to define when a destination creation pair P in in the set of R_{join} (rule `RJOIN`) or R_{branch} (rule `RBRANCH`).

Definition 10 (Root of the joined destination creation set) *To construct the destination document from the set of joined destination creation pairs in R_{join}, the first component x referenced in each destination creation pair P must be the same (because they are to be joined). We write the following to locate the element x:*

$$R_{join} \rhd x$$

Rule `ROOT.RJOIN` in Figure 5.16 defines the path component x located in the set of joined destination creation pairs R_{join}.

Definition 11 (Create Sequence) *During the creation of the destination typed document, it is necessary to combine typed documents to form a sequence. To*

$$\texttt{RBRANCH}\quad \frac{P \in R \quad P = [\delta \times td] \quad \delta = \langle [\theta \times branch], \ldots \rangle}{P \in R_{branch}}$$

$$\texttt{RJOIN}\quad \frac{P \in R \quad P = [\delta \times td] \quad \delta = \langle [\theta \times join], \ldots \rangle}{P \in R_{join}}$$

$$\texttt{ROOT.RJOIN}\quad \frac{\begin{array}{c} R_{join} = \{P_1, P_2, \ \ldots \ , P_n\} \\ P_1 = [\rho_1, td_1] \quad \rho_1 = \langle [x \times join], \ \ldots \ \rangle, \\ P_2 = [\rho_2, td_2] \quad \rho_2 = \langle [x \times join], \ \ldots \ \rangle, \\ \ldots \ , \\ P_n = [\rho_n, td_n] \quad \rho_n = \langle [x \times join], \ \ldots \ \rangle \end{array}}{R_{join} \rhd x}$$

FIGURE 5.16: Rules to define the sets of joined and branched destination creation pairs.

$$\texttt{MAKE.SEQA}\quad \frac{td_a \neq \epsilon \wedge td_b = \epsilon}{td_a \sqcap td_b = td_a}$$

$$\texttt{MAKE.SEQB}\quad \frac{td_a = \epsilon \wedge td_b \neq \epsilon}{td_a \sqcap td_b = td_b}$$

$$\texttt{MAKE.SEQAB}\quad \frac{td_a \neq \epsilon \wedge td_b \neq \epsilon}{td_a \sqcap td_b = td_a, td_b}$$

FIGURE 5.17: Rules to define the construction of sequences.

combine td_a and td_b we write:

$$td_a \sqcap td_b = td_r$$

Figure 5.17 contains three rules to define the creation of a sequence from two documents td_a and td_b. Rule `MAKE.SEQA` is used when td_b is equal to the empty document (ϵ), so $td_a \sqcap td_b = td_a$. Rule `MAKE.SEQB` is used when td_a is equal to the emtpy document (ϵ), so $td_a \sqcap td_b = td_b$. Finally, when both td_a and td_n are not equal to the empty document, $td_a \sqcap td_b$ is equal to a typed document that is the sequence td_a, td_b.

Definition 12 (Destination Document Construction) *When mappings have been applied to a source document to make the set of destination creation pairs R*

(where $R = \{P_1, P_2, \ldots, P_n\}$ and $P_n = [\delta \times td_c]$), R can be used to construct the destination document td_r using ***Definition 8****. To denote this we write:*

$$\mathbf{construct}(R) = td_r$$

Figure 5.18 contains rules to define the construction of documents using the set of destination creation pairs R. Rule `R.EVAL` states the set R is divided into two subsets called R_{join} and R_{branch} that are used to construct two result documents td_j and td_b. Therefore, the construction of a destination document using R is equal to the combination of td_b and td_b (see previous rules in Figure 5.17).

Rule `RJOIN.EVAL` defines the construction of a destination document using the set R_{join}. Each destination creation pair P_i has the first destination mapping pair removed to give P'_i ($\mathbf{next}(P_i) = P'_i$ using rule `NEXT.C.PAIR`). These new destination content pairs are then combined in the set R' which is itself used to construct the result document td_r. The root element x is located ($R_{join} \rhd x$), and its type is determined ($\mathbf{deref}(x).type = t$) so the destination document $x[t \ni td_r]$ can be created.

Rule `RBRANCH.EVAL` defines the construction of a destination document using the set R_{join}. Each destination creation pair $P_n \in R_{branch}$ is used to construct a destination document td_n using rules `BPAIR.EVAL.E`, `BPAIR.EVAL.A`, `BPAIR.EVAL.C`, or `BPAIR.EVA.EMP` (defined earlier in Figure 5.15) if the destination mapping path δ contains only one pair, or rule `BPAIR.EVAL.LIST` if there is more than one pair in the destination mapping path. Rule `BPAIR.EVAL.LIST` defines the construction of a destination document using a destination creation pair P that contains a destination mapping path δ with more than one pair. The first component referenced (x) and its type (t) are determined, and the destination creation pair P has its first destination mapping pair removed to give P' (written $\mathbf{next}(P) = P'$). A set of new destination creation pairs R is created that contains only P'. R is then used to construct the destination document td_r (with rule `R.EVAL`), and therefore P constructs the document $x[t \ni td_r]$.

Definition 13 (Mapping Application) *The evaluation of a mapping m from the binding B against a typed document td_s gives a destination creation pair P where $P = [\delta \times td_r]$. The typed document td_r is the result of the application of the*

$$\texttt{R.EVAL}\quad \frac{R = R_{join} \cup R_{branch} \qquad \mathbf{construct}(R_{join}) = td_j \quad \mathbf{construct}(R_{branch}) = td_b}{\mathbf{construct}(R) = td_j \sqcap td_b}$$

$$\texttt{RJOIN.EVAL}\quad \frac{\begin{array}{c} R_{join} = \{P_1, P_2, \ \dots\ , P_i\} \\ \mathbf{next}(P_1) = P_1', \ \mathbf{next}(P_2) = P_2', \ \dots\ , \mathbf{next}(P_i) = P_i' \\ R' = \{P_1', P_2', \ \dots\ , P_i'\} \\ \mathbf{construct}(R') = td_r \\ R_{join} \rhd x \quad \mathbf{deref}(x).type = t \end{array}}{\mathbf{construct}(R_{join}) = x[t \ni td_r]}$$

$$\texttt{RBRANCH.EVAL}\quad \frac{\begin{array}{c} R_{branch} = \{P_1, P_2, \ \dots\ , P_k\} \\ \mathbf{construct}(P_1) = td_1, \\ \mathbf{construct}(P_2) = td_2, \\ \dots\ , \\ \mathbf{construct}(P_n) = td_n \end{array}}{\mathbf{construct}(R_{branch}) = td_1 \sqcap td_2 \sqcap \ \dots\ \sqcap td_n}$$

$$\texttt{NEXT.C.PAIR}\quad \frac{\begin{array}{c} P = [\delta, td_s] \\ \delta = \langle [\theta_h \times \omega_h], [\theta_r \times \omega_r], \ \dots \rangle \\ \delta_{rest} = \langle [\theta_r \times \theta_r], \ \dots \rangle \end{array}}{\mathbf{next}(P) = [\delta_{rest} \times td_s]}$$

$$\texttt{BPAIR.EVAL.LIST}\quad \frac{\begin{array}{c} P = [\delta \times td_s] \\ \delta = \langle [x \times branch], [\theta_r \times \omega_r], \ \dots \rangle \\ \mathbf{deref}(x)\texttt{.type} = t \\ \mathbf{next}(P) = P' \\ R = \{P'\} \\ \mathbf{construct}(R) = td_r \end{array}}{\mathbf{construct}(P) = x[t \ni td_r]}$$

Figure 5.18: Rules to define the construction of the destination document.

source mapping path ρ from m to td_s, and δ is the destination mapping path:

$$m, B \vdash td_s \rightarrow [\delta \times td_r]$$

Because more than one mapping may be applied to a given typed document, we define the application of a set of applicable mappings M_a to a typed document td_s as a set of result pairs R where $R = \{P_1, P_2, \ \ldots \ , P_n\}$:

$$\mathbf{evaluate}(M_a, td_s) = R$$

Rules for the application of mappings are given in Figure 5.19. Rule `MAP.EVAL` states that when the mapping m in B is valid for application to a source typed document td_s, the result of the application of ρ to td_s is td_r. Local mappings B_l are combined with the global binding B to give B' (where ordering is preserved) that is used to transform the result document td_r into $td_{r'}$. The result of the recursion ($td_{r'}$) is then combined with the destination mapping path δ to give the destination creation pair $[\delta \times td_{r'}]$.

Rule `MAPSET.EVAL` describes how a set of compatible mappings M_a are each evaluated against a source document td_s to give the set of result pairs R where $R = \{P_1, P_2, \ \ldots \ , P_n\}$.

Definition 14 (Document Transformation) *The transformation of a source document td_s using mappings from the binding B creates a destination document td_r and is denoted by:*

$$\mathbf{transform}(B, td_s) = td_r$$

Rule `BINDING.EVAL` in Figure 5.19 defines this behaviour. The set of compatible mappings M_a is calculated and evaluated to give a set of destination creation pairs R ($\mathbf{evaluate}(M_a, td_s) = R$). R is then used to construct the destination td_r ($\mathbf{construct}(R) = td_r$) — the result of the transformation process.

$$\texttt{MAP.EVAL}\quad \frac{\begin{array}{c} m \in B \\ m = \langle \rho,\ \delta,\ B_l \rangle \\ \textbf{isCompatible}(m, td_s) \\ \rho \vdash td_s \rightarrow td_r \\ B' = B \cup B_l \\ \textbf{transform}(B', td_r) = td_{r'} \end{array}}{m, B \vdash td_s \rightarrow [\delta \times td_{r'}]}$$

$$\texttt{MAPSET.EVAL}\quad \frac{\begin{array}{c} td_s \quad B \\ M_a = \langle m_1, m_2,\ \dots\ , m_n \rangle \\ \textbf{isCompatible}(m_1, td_s), \\ \textbf{isCompatible}(m_2, td_s), \\ \dots \\ \textbf{isCompatible}(m_n, td_s) \\ m_1 \in B, m_2 \in B,\ \dots\ , m_n \in B \\ m_1, B \vdash td_s \rightarrow P_1,\ m_2, B \vdash td_s \rightarrow P_2,\ \dots,\ m_n, B \vdash td_s \rightarrow P_n \end{array}}{\textbf{evaluate}(M_a, td_s) = \{P_1,\ P_2,\ \dots\ ,\ P_n\}}$$

$$\texttt{BINDING.EVAL}\quad \frac{\begin{array}{c} B \quad td_s \\ M_a = \langle m_1,\ m_2,\ \dots\ , m_n \rangle \\ \textbf{isCompatible}(m_1, td_s), \\ \textbf{isCompatible}(m_2, td_s), \\ \dots \\ \textbf{isCompatible}(m_n, td_s) \\ m_1 \in B, m_2 \in B,\ \dots\ , m_n \in B \\ \textbf{evaluate}(M_a, td_s) = R \\ \textbf{construct}(R) = td_r \end{array}}{\textbf{transform}(B, td_s) = td_r}$$

FIGURE 5.19: Rules to define the evaluation of Bindings.

5.6 Example Mappings

To demonstrate our mapping language, we provide a subset of mappings to transform an instance of a DDBJ sequence data record to a *Sequence_Data_Record* concept instance (the full set of mappings can be found in in Appendix B). For compactness, assume all source mapping path predicates are *true* unless otherwise specified (see mapping 12 and 14):

$$\begin{aligned}
m_1 &= \langle\ \langle \text{DDBJXML}, \text{ACCESSION}\rangle\ ,\ \langle[\text{Sequence\_Data\_Record} \times \textit{join}], [\text{accession\_id} \times \textit{branch}]\rangle\ , \emptyset\rangle \\
m_2 &= \langle\ \langle \text{ACCESSION}, \textit{value}\rangle\ ,\ \langle[\text{accession\_id} \times \textit{join}], \textit{value}\rangle\ , \emptyset\rangle \\
m_3 &= \langle\ \langle \text{DDBJXML}, \text{DEFINITION}\rangle\ ,\ \langle[\text{Sequence\_Data\_Record} \times \textit{join}], [\text{definition} \times \textit{branch}]\rangle\ , \emptyset\rangle \\
m_4 &= \langle\ \langle \text{DEFINITION}, \textit{value}\rangle\ ,\ \langle[\text{definition} \times \textit{join}], \textit{value}\rangle\ , \emptyset\rangle \\
m_7 &= \langle\ \langle \text{source}, \text{location}\rangle\ ,\ \langle[\text{Feature\_Source} \times \textit{join}], [\text{has\_position} \times \textit{branch}], [\text{Location} \times \textit{branch}]\rangle\ , \emptyset\rangle \\
m_9 &= \langle\ \langle \text{location}, \textit{value}\{\text{``\^{}[\^{}.]+''}\}\rangle\ ,\ \langle[\text{Location} \times \textit{join}], [\text{start} \times \textit{branch}], \textit{value}\rangle\ , \emptyset\rangle \\
m_{10} &= \langle\ \langle \text{location}, \textit{value}\{\text{``[\^{}.]+''}\}\rangle\ ,\ \langle\text{Location} \times \textit{join}], [\text{end} \times \textit{branch}], \textit{value}\rangle\ , \emptyset\rangle \\
m_{11} &= \langle\ \langle \text{DDBJXML}, \text{FEATURES}, \text{source}\rangle\ , \\
&\quad \langle[\text{Sequence\_Data\_Record} \times \textit{join}], [\text{has\_feature} \times \textit{branch}], [\text{Feature\_Source} \times \textit{branch}]\rangle\ , \emptyset\rangle \\
m_{12} &= \langle\ \langle \text{source}, [\text{qualifiers} \times \{\text{qualifiers}, \text{qualifiers/*/@name}\textit{value} = \text{``isolate''}\}]\rangle\ , \\
&\quad \langle[\text{Feature\_Source} \times \textit{join}], [\text{isolate} \times \textit{branch}]\rangle\ ,\ (m_{13})\rangle \\
m_{13} &= \langle\ \langle \text{qualifiers}, \textit{value}\rangle\ ,\ \langle[\text{isolate} \times \textit{join}], \textit{value}\rangle\ , \emptyset\rangle \\
m_{14} &= \langle\ \langle \text{source}, [\text{qualifiers} \times \{\text{qualifiers}, \text{qualifiers/*/@name}\textit{value} = \text{``lab\_host''}\}]\rangle\ , \\
&\quad \langle[\text{Feature\_Source} \times \textit{join}], [\text{lab\_host} \times \textit{branch}\}]\rangle\ , (m_{15})\rangle \\
m_{15} &= \langle\ \langle \text{qualifiers}, \textit{value}\rangle\ ,\ \langle[\text{lab\_host} \times \textit{join}], \textit{value}\rangle\ , \emptyset\rangle
\end{aligned}$$

These mappings are then used to define a binding B as follows:

$$B = \langle m_1, m_2, m_3, m_4, m_7, m_9, m_{10}, m_{11}, m_{12}, m_{14}\rangle \tag{5.1}$$

Mappings m_{13} and m_{15} are excluded from the sequence B because they are defined locally within other mappings. A source document in DDBJ format can then be evaluated using this binding to give a destination document which is the sequence data record in its corresponding OWL representation.

5.7 XML Syntax for Binding Specification

The specification of mappings and bindings in XML format is supported, as we illustrate in Listing 5.4, where an equivalent binding is given to the one specified in Section 5.6. Mapping ids are consistent so the reader can easily find the corresponding mapping in mathematical notation. This kind of XML document is called an M-Binding and can be used to drive the translation of XML documents, as we show later in Chapter 6. Our XML binding format is designed to look similar to conventional XPATH notation so users familiar with XML tools will find it intuitive.

```
<binding name="DDBJ–to–sequencedata"
         xmlns="http://jaco.ecs.soton.ac.uk/schema/binding"
         xmlns:sns="http://jaco.ecs.soton.ac.uk/schema/DDBJ"
         xmlns:dns="http://jaco.ecs.soton.ac.uk/ont/sequencedata"
         targetNamespace="http://jaco.ecs.soton.ac.uk/binding/DDBJ–to–sequencedata">
  <mapping id=’m1’>
    <source match="sns:DDBJXML/sns:ACCESSION"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:accession_id[branch]/"/>
  </mapping>

  <mapping id=’m2’>
      <source match="sns:ACCESSION/$"/>
      <destination create="dns:accession_id[join]/$"/>
  </mapping>

  <mapping id=’m3’>
    <source match="sns:DDBJXML/sns:DEFINITION"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:definition[branch]/"/>
  </mapping>

  <mapping id=’m4’>
    <source match="sns:DEFINITION/$"/>
    <destination create="dns:definition[join]/$"/>
  </mapping>

  <mapping id=’m7’>
    <source match="sns:source/sns:location"/>
    <destination create="dns:Feature_Source[join]/dns:has_position[branch]/dns:Location[branch]"/>
  </mapping>

  <mapping id=’m9’>
    <source match="sns:location/$^[^.]+"/>
    <destination create="dns:Location[join]/dns:start[branch]/$"/>
  </mapping>

  <mapping id=’m10’>
    <source match="sns:location/$[^.]+$"/>
    <destination create="dns:Location[join]/dns:end[branch]/$"/>
  </mapping>

  <mapping id=’m11’>
    <source match="sns:DDBJXML/sns:FEATURES/sns:source"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:has_feature[branch]/dns:Feature_Source[branch]"/>
  </mapping>

  <mapping id=’m12’>
    <source match=’sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "isolate"]’/>
    <destination create="dns:Feature_Source[join]/dns:isolate[branch]"/>
    <mapping id=’m13’>
      <source match="sns:qualifiers/$"/>
      <destination create="dns:isolate[join]/$"/>
    </mapping>
  </mapping>

  <mapping id=’m14’>
    <source match=’sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "lab_host"]’/>
    <destination create="dns:Feature_Source[join]/dns:lab–host[branch]"/>
    <mapping id=’m15’>
      <source match="sns:qualifiers/$"/>
      <destination create="dns:lab–host[join]/$"/>
    </mapping>
  </mapping>
</binding>
```

Listing 5.4: An XML representation for a Binding

Local mappings can be defined easily by including their definition within the parent mapping element (see mappings 12 and 14). To extract literal values from the content of an element or attribute, the $ symbol is used, and can be suffixed with a string to denote a regular expression (mappings 9 and 10).

5.8 Conclusions

The mapping and transformation formalism presented in this Chapter provides an XML to XML transformation technology based on the MSL formalisation of XML and XML schema. While we use this language to describe the conversion of an XML document to, and from, a canonical OWL serialisation, the formalism can be used as a generic XML to XML translation tool. Mapping statements describe the association of XML components (elements, attributes and literal values) in a source schema to components in a destination schema, so such mappings can be used to drive the transformation of a source document. The following advanced mapping constructs are supported:

- **Document paths**
  Simple transformations can be expressed using 1 *to* 1 mappings. To accommodate scenarios where a single component maps to a set of components (1 *to* n), or a set of components map to a single component (n *to* 1), mapping statements can be expressed using document paths. For example, m_{11} from the example mapping in Section 5.6 maps `DDBJXML/FEATURES/source` to `Sequence_Data_Record` .

- **Predicate support**
  When the mapping of a component is dependent on the value of another attribute or element, such as the `<qualifiers>` element in the DDBJ sequence data record, predicate evaluation is used - see m_{12}. In this example, the value of the @name attribute must be "isolate" for the `<qualifiers>` element to be mapped to the `<isolate>` element.

- **Scoping**
  Sometimes the mapping of a particular element or attribute depends on context. For example, the value of the `<qualifiers>` element is mapped differently in mappings m_{13} (local to mapping m_{12}), and m_{15} (local to m_{14}).

- **String Manipulation**
  When the value of an element contains two distinct entities, such as the `<location>` element in the DDBJ record, regular expressions can be used to extract different characters from an elements content. An example of this construct can be found in mappings m_9 and m_{10}.

The translation process is a recursion over the source document that applies compatible mappings at each element or attribute encountered to create elements and attributes in the destination document. By using a modular specification approach we facilitate the reuse of mappings when service operations are defined across the same or subsets of the same XML schema.

One mapping construct not supported is *list processing.* Within XML schema, elements can contain sequences of other elements. Although it is not necessary to meet the requirements from our bioinformatics data set, it would be desirable to add mapping constructs that enable elements within a sequence to be mapped differently depending on their position. For example, map the first instance to one element and the rest to another. This is supported in XPATH where array indexes can be used, for example, `a/b[0]` will return the first `<b>` element contained within `<a>` .

While the use of the joining operator is critical to our translation formalism, it is also cumbersome. By analysing the destination schema to see what destination documents are valid, the user could be freed of this burden.

Chapter 6

The Configurable Mediator Implementation

To enable a client within the WS-HARMONY architecture to perform workflow harmonisation, the Configurable Mediator (introduced in Chapter 4) can be used to create a Type Adaptor on-the-fly by consuming the appropriate realisation and serialisation mappings. In Chapter 5, an XML mapping and transformation formalism (FXML-M) was presented to enable complex mappings to be made between XML schema components that can be used to drive the transformation of a document. In this Chapter, we present an implementation of this formalisation in the form of a SCHEME [63] library called FXML-T (Formalised **XML** Translation) which offers the following functionality:

1. A SCHEME representation for MSL [26] components and typed documents.
2. A number of functions to import conventional XML documents, XML schemas and M-Binding documents into FXML-T s-expressions.
3. A SCHEME representation for mappings and M-Bindings, supporting document paths and predicate evaluation.
4. Functions to perform document translation using an M-Binding according to the rules presented earlier in Chapter 5.

This SCHEME library is used to construct a *Translation Engine* which is combined with the JENA ontology processing API to create the Configurable Mediator. To evaluate the practicality and scalability of our mapping language implementation, as well as examine the relative cost of composing M-Bindings, we test the FXML-T library using increasing document sizes, increasing schema sizes, increasingly complex M-Binding composition, and real bioinformatics data. Evaluation shows our implementation scales well and M-Binding composition comes with virtually zero performance cost. We also examine the complexity of our transformation algorithm and show that translation cost is $O(c, n)$ where c is the number of compatible mappings, and n is the size of the input document. Hence, the contribution of this Chapter is the Configurable Mediator: An efficient software component that is dynamically configured by realisation and serialisation M-Bindings to create intermediary-based Type Adaptors.

We begin this Chapter in Section 6.1 with a brief discussion of macro languages and our implementation of FXML-M relates to these. Section 6.2 provides the FXML-T representation of normalised schema names, schema components, typed documents and mappings, providing example SCHEME code to illustrate their representation. Section 6.3, contains definitions of the functions offered by the FXML-T library to enable the conversion of XML documents, XML schemas and M-Binding documents to FXML-T, as well as the transformation of documents using M-Bindings. Section 6.3.2 presents pseudocode for our transformation algorithm and an analysis of its complexity. We then show how these functions can be combined to provide a Transformation Engine in Section 6.3, before presenting the internal workings of the Configurable Mediator in Section 6.4. Section 6.5 gives details of our evaluation including a comparison with other XML translation technologies. Finally, we conclude the Chapter in Section 6.6.

6.1 Transformation Languages

Transformation languages have been studied within the computer science discipline [67] since the 1960s when the first programming languages were developed. When using the first generation of computers, programmers were limited to writing code using assembly languages where instructions in the source program have a one-to-one correspondence to the instructions executed by the central processing unit. Programmers realised early on that much of the code written was duplicated so some simple reuse mechanisms were introduced so that symbols could be used to denote the inclusion of a large block of code. As these reuse mechanisms matured, facilities were added to include different code based on the value of some parameters, and the first macro languages we conceived, including the General-Purpose Macro Processor (GPM) [88], Macro Language One (ML/1) [28], and TRAC [75].

As programming languages and compiler engineering advanced, the requirements for macro languages became more complex. It was recognised that simple text rewriting was not sufficient when trying to express intricate transformations of data and programs. To overcome these limitations, tree rewriting systems were developed, giving programmers the means to express elaborate data transformations. A good example of this is the R5RS SCHEME [63] macro system that has two notable features:

1. **Hygienic Macros**
   When a macro is expanded, the system automatically creates private symbols that bind to the macro parameters. This avoids the problem of *variable capture* where statements in the macro expression share names with variables already in the environment, resulting in unexpected behaviour.

2. **Pattern Matching**
   Instead of using SCHEME code to define pattern matching, a declarative syntax is provided to give programmers a more intuitive interface to the macro system.

In addition, there is a vast amount of literature providing theoretical foundations to tree rewriting system, an example of which is the Lambda Calculus [15].

When implementing the FXML-M language, we are essentially creating a tree rewriting system; the input is an XML document (a tree structure) and the output is a different XML document. Like the SCHEME macro system, we use a declarative approach for pattern matching to maximise accessibility. Since XML pattern matching is already widely used in the community through XPATH [35], we ensure that our pattern matching syntax is close to XPATH. For the implementation of our transformation algorithm, we choose SCHEME because: (1) the similarity between XML tree structures to SCHEME s-expressions means there is little overhead to model XML documents (2) as we show in later in Section 6.3.1, the transformation rules defined formally in Chapter 5 can be easily specified in SCHEME.

6.2 FXML-T Representation Overview

In this Section, we describe how aspects of MSL and our mapping formalisation (FXML-M) are represented in FXML-T. We show the format of normalised component names, schema components, typed documents and mappings, providing example SCHEME s-expressions to illustrate their representation.

6.2.1 FXML-T representation of normalised component names

In MSL, normalised component names are used to reference elements, attributes and types. We define the structure of a normalised component name in FXML-T using BNF [13] notation:

$$
\begin{array}{rcl}
\langle fxml : cname \rangle & ::= & (\ \langle uri \rangle\ .\ \langle localname* \rangle\ ) \\
\langle uri \rangle & ::= & \langle string \rangle \\
\langle localname* \rangle & ::= & \langle\ \rangle \mid (\ \langle localname \rangle\ .\ \langle localname* \rangle\ ) \\
\langle localname \rangle & ::= & (\ \langle sort \rangle\ .\ \langle string \rangle\ ) \\
\langle sort \rangle & ::= & \textbf{element} \mid \textbf{attribute} \mid \textbf{complexType} \mid \textbf{simpleType} \\
\langle string \rangle & ::= & \text{“}\langle sequence\ of\ characters \rangle\text{”}
\end{array}
$$

An $\langle fxml : cname \rangle$ is a pair containing a namespace URI and a list of localnames. Each localname is a pair containing the component sort and component name. We provide an example in Figure 6.1 that gives a component name, in MSL notation, and the corresponding SCHEME s-expression to represent it. The bounding box illustrates where the namespace `"http://jaco.ecs.soton.ac.uk/schema/DDBJ"` appears in both representations.

6.2.2 FXML-T representation of schema components

In MSL, components describe the elements, attributes and types of an XML schema. An XML schema is represented in FXML-T using an $\langle fxml : schema \rangle$, defined as

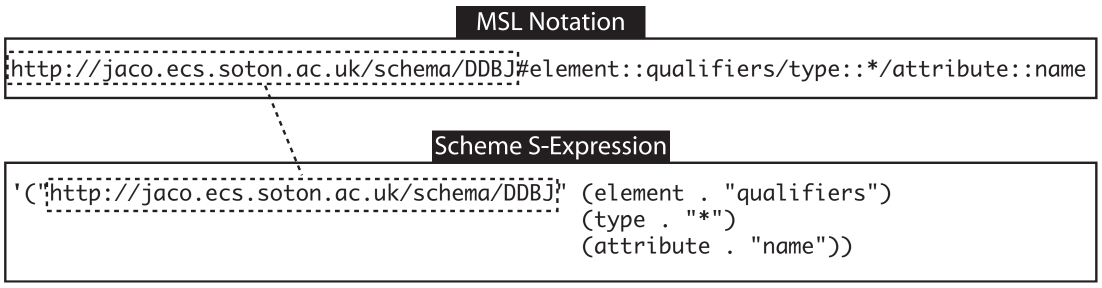

MSL Notation
http://jaco.ecs.soton.ac.uk/schema/DDBJ#element::qualifiers/type::*/attribute::name
Scheme S-Expression
'("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers")
(type . "*")
(attribute . "name"))

FIGURE 6.1: Component Name representation in FXML-T

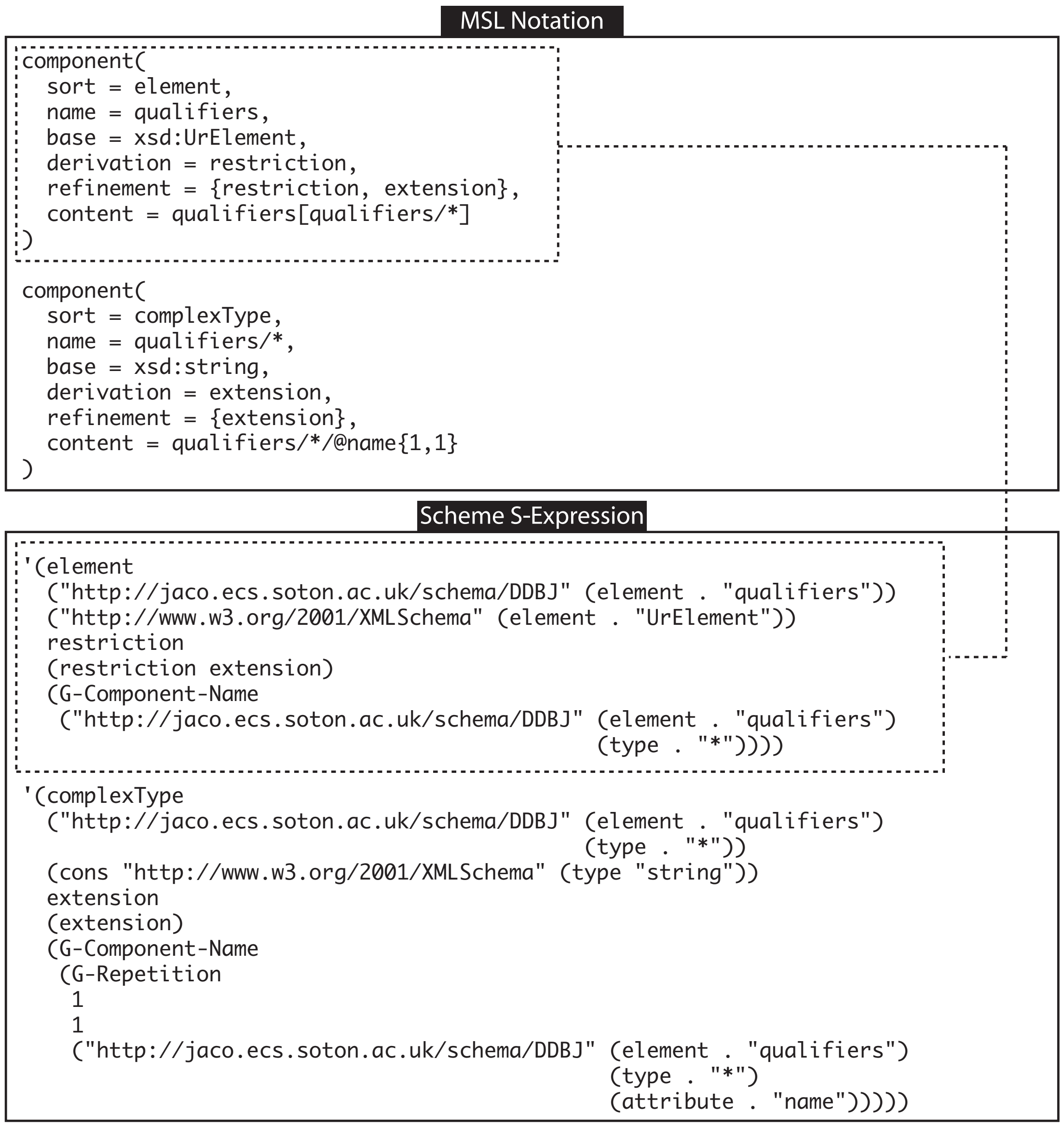
follows:

$$
\begin{array}{rcl}
\langle fxml : schema\rangle & ::= & \langle list\, of\, fxml : component\rangle \\
\langle fxml : component\rangle & ::= & (\ \langle sort\rangle \\
 & & \ \langle fxml : cname\rangle \\
 & & \ \langle base\rangle \\
 & & \ (\ \langle derivation\rangle\ ) \\
 & & \ \langle refinement\rangle \\
 & & \ \langle content\rangle\ ) \\
\langle base\rangle & ::= & \langle fxml : cname\rangle \\
\langle derivation\rangle & ::= & \textbf{restriction} \mid \textbf{extension} \\
\langle refinement\rangle & ::= & \langle list\ of\ derivation\rangle \\
\langle content\rangle & ::= & (\ \textbf{G-Sequence}\ \langle content*\rangle\ ) \\
 & \mid & (\ \textbf{G-Choice}\ \langle content*\rangle\ ) \\
 & \mid & (\ \textbf{G-Interleave}\ \langle content*\rangle\ ) \\
 & \mid & (\ \textbf{G-Repetition}\ \langle min\rangle\langle max\rangle\langle content\rangle\ ) \\
 & \mid & (\ \textbf{G-Attribute}\ \langle fxml : cname\rangle\langle content\rangle\ ) \\
 & \mid & (\ \textbf{G-Element}\ \langle fxml : cname\rangle\langle content\rangle\ ) \\
 & \mid & (\ \textbf{G-Component-Name}\ \langle fxml : cname\rangle\ ) \\
\langle content*\rangle & ::= & \langle list\ of\ content\rangle \\
\langle min\rangle & ::= & \langle integer\rangle \\
\langle max\rangle & ::= & \langle integer\rangle \mid \textbf{infinite}
\end{array}
$$

An $\langle fxml : component\rangle$ is a list containing a sort[1], the name of the component, the name of the base component, the derivation type, the permitted refinements and the content. To illustrate the $\langle fxml : component\rangle$ representation, Figure 6.2 gives example SCHEME s-expressions to create two components from the DDBJ sequence data schema. The `qualifiers` element contains string content and has exactly one

[1]The term *sort* is used to avoid confusion with the XML term *type*.

attribute called `name`. Using this notation, an XML schema is represented in FXML-T as a list of $\langle fxml : component \rangle$.



MSL Notation

```
component(
  sort = element,
  name = qualifiers,
  base = xsd:UrElement,
  derivation = restriction,
  refinement = {restriction, extension},
  content = qualifiers[qualifiers/*]
)

component(
  sort = complexType,
  name = qualifiers/*,
  base = xsd:string,
  derivation = extension,
  refinement = {extension},
  content = qualifiers/*/@name{1,1}
)
```

Scheme S-Expression

```
'(element
  ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers"))
  ("http://www.w3.org/2001/XMLSchema" (element . "UrElement"))
  restriction
  (restriction extension)
  (G-Component-Name
   ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers")
                                              (type . "*"))))
'(complexType
  ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers")
                                             (type . "*"))
  (cons "http://www.w3.org/2001/XMLSchema" (type "string"))
  extension
  (extension)
  (G-Component-Name
   (G-Repetition
    1
    1
    ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers")
                                               (type . "*")
                                               (attribute . "name")))))
```

FIGURE 6.2: Component representation in FXML-T

6.2.3 FXML-T representation of typed documents

In MSL, XML documents are formed using typed documents. In FXML-T, typed documents are defined as follows:

$$\begin{aligned}
\langle fxml : td \rangle \quad &::= \quad (\ \textbf{TD-Empty}\ ) \\
&\quad\ \mid \quad (\ \textbf{TD-Constant}\ \langle \mathit{string} \rangle\ ) \\
&\quad\ \mid \quad (\ \textbf{TD-Sequence}\ \langle fxml : td \rangle \langle fxml : td \rangle\ ) \\
&\quad\ \mid \quad (\ \textbf{TD-Element}\ \langle \mathit{name} \rangle \langle \mathit{type} \rangle \langle fxml : td \rangle\ ) \\
&\quad\ \mid \quad (\ \textbf{TD-Attribute}\ \langle \mathit{name} \rangle \langle \mathit{type} \rangle \langle fxml : td \rangle\ ) \\
\langle \mathit{name} \rangle \quad &::= \quad \langle fxml : \mathit{cname} \rangle \\
\langle \mathit{type} \rangle \quad &::= \quad \langle fxml : \mathit{cname} \rangle
\end{aligned}$$

An $\langle fxml : td \rangle$ is one of five sorts: The *empty document* (for empty XML elements), a *constant value* (e.g. a string literal or integer value), a *sequence* containing two typed documents (for elements containing other elements), an *element* (with a name and type) containing a typed document, or an *attribute*(with a name and type) containing a typed document. We give an example SCHEME s-expression in Figure 6.3 to create a small DDBJ sequence data document. The `<DDBJXML>` element contains a sequence of two typed documents holding the `<ACCESSION>` and `<SEQUENCE>` elements, each having string content.

MSL Notation

```
DDBJXML[DDBJXML/* ∋
  ACCESSION[xsd:string ∋ "AB000059"],
  SEQUENCE[xsd:string ∋ "atgagtgatggagcagttcaaccagacgg..."]
]
```

Scheme Code

```
'(TD-Element
  ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "DDBJXML"))
  ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "DDBJXML")
                                             (type . "*"))
  (TD-Sequence
   (TD-Element
     ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "ACCESSION"))
     ("http://www.w3.org/2001/XMLSchema" (type . "string"))
     (TD-Constant "AB000059"))
    (TD-Element
     ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "SEQUENCE"))
     ("http://www.w3.org/2001/XMLSchema" (type . "string"))
     (TD-Constant "atgagtgatggagcagttcaaccagacgg..."))))
```

FIGURE 6.3: Typed document representation in FXML-T

6.2.4 FXML-T Representation of bindings and mappings

In Chapter 5, a binding is defined as a sequence of mappings where a binding may be represented in XML format as an M-Binding document. Each mapping specifies the relation between XML components in a source schema to components in a destination schema. Such a binding can then be used to direct the translation of an XML document to a different representation. In FXML-T, bindings and mappings are defined as follows:

$$
\begin{array}{rcl}
\langle fxml : binding\rangle & ::= & \langle\,\rangle \mid \textbf{(}\ \langle fxml : mapping\rangle\ .\ \langle fxml : binding\rangle\ \textbf{)} \\
\langle fxml : mapping\rangle & ::= & \textbf{(}\ \langle id\rangle\langle scope\rangle\langle spairs{*}\rangle\langle dpairs{*}\rangle\langle local\rangle\ \textbf{)} \\
\langle id\rangle & ::= & \langle string\rangle \\
\langle scope\rangle & ::= & \textbf{global} \mid \textbf{local} \\
\langle spairs{*}\rangle & ::= & \langle\,\rangle \mid \textbf{(}\ \langle spair\rangle\ .\ \langle spair{*}\rangle\ \textbf{)} \\
\langle spair\rangle & ::= & \textbf{(}\ \langle spath\rangle\ .\ \langle fxml : predicate\rangle\ \textbf{)} \\
\langle spath\rangle & ::= & \langle fxml : cname\rangle \mid \textbf{(}\ \textbf{empty}\ \textbf{)} \mid \langle svalue\rangle \\
\langle svalue\rangle & ::= & \textbf{(}\ \textbf{value}\ \langle regexp\rangle\ \textbf{)} \mid \textbf{(}\ \textbf{value}\ \textbf{)} \\
\langle regexp\rangle & ::= & \langle string\rangle \\
\langle fxml : predicate\rangle & ::= & \textbf{(}\ \textbf{true}\ \textbf{)} \\
 & \mid & \textbf{(}\ \textbf{exists}\ \langle fxml : pexpr{*}\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ \textbf{not}\ \langle fxml : pexpr{*}\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ \textbf{and}\ \langle fxml : predicate\rangle\langle fxml : predicate\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ \textbf{or}\ \langle fxml : predicate\rangle\langle fxml : predicate\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ =\ \langle fxml : pexpr{*}\rangle\langle fxml : pexpr{*}\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ >\ \langle fxml : pexpr{*}\rangle\langle fxml : pexpr{*}\rangle\ \textbf{)} \\
 & \mid & \textbf{(}\ <\ \langle fxml : pexpr{*}\rangle\langle fxml : pexpr{*}\rangle\ \textbf{)}
\end{array}
$$

$$
\begin{aligned}
\langle \textit{pexpr}* \rangle &::= \langle\,\rangle \mid \textbf{(}\ \langle \textit{pexpr} \rangle\ .\ \langle \textit{pexpr}* \rangle\ \textbf{)} \\
\langle \textit{pexpr} \rangle &::= \langle \textit{fxml} : \textit{cname} \rangle \mid \langle \textit{constant} \rangle \mid \textbf{value} \\
\langle \textit{dpairs}* \rangle &::= \langle\,\rangle \mid \textbf{(}\ \langle \textit{dpair} \rangle\ .\ \langle \textit{dpair}* \rangle\ \textbf{)} \\
\langle \textit{dpair} \rangle &::= \textbf{(}\ \langle \textit{dpath} \rangle\ .\ \langle \textit{operator} \rangle\ \textbf{)} \\
\langle \textit{dpath} \rangle &::= \langle \textit{fxml} : \textit{cname} \rangle \mid \textbf{(value)} \mid \textbf{(empty)} \\
\langle \textit{operator} \rangle &::= \textbf{(branch)} \mid \textbf{(join)} \\
\langle \textit{local} \rangle &::= \langle\,\rangle \mid \textbf{(}\ \langle \textit{fxml} : \textit{mapping} \rangle\ .\ \langle \textit{local} \rangle\ \textbf{)} \\
\langle \textit{constant} \rangle &::= \langle \textit{string} \rangle
\end{aligned}
$$

An $\langle fxml$: *binding*$\rangle$ is a list of mappings. Each $\langle fxml$: *mapping*$\rangle$ is a list containing an identifier, the scope of the mapping (either local or global), a source mapping path ($\langle$*spairs*$*\rangle$), a destination mapping path ($\langle$*dpairs*$*\rangle$), and a list of local mappings. A $\langle$*spair*$\rangle$ is a pair containing a source path and a predicate. A $\langle$*spath*$\rangle$ is either an $\langle fxml$: *cname*$\rangle$, the keyword *empty*, or an $\langle$*svalue*$\rangle$ expression which can include a regular expression to extract particular characters from a string value. An $\langle fxml$: *predicate*$\rangle$ can be one of eight sorts: *true*, *exists*, *not*, *and*, *or*, =, <, or >. A $\langle$*dpair*$\rangle$ is a pair containing a destination path and a joining operator. To demonstrate the construction of a binding in FXML-T, Figure 6.4 contains a SCHEME s-expression to create a subset of mappings that describe the translation of a DDBJ document to a `Sequence_Data_Record` concept instance (The full set of mappings can be found in Appendix B).

$$
\begin{aligned}
m_1 &= \langle\ \langle \text{DDBJXML}, \text{ACCESSION} \rangle\ ,\ \langle [\text{Sequence\_Data\_Record} \times \textit{join}], [\text{accession\_id} \times \textit{branch}] \rangle\ , \emptyset \rangle \\
m_2 &= \langle\ \langle \text{ACCESSION}, \textit{value} \rangle\ ,\ \langle [\text{accession\_id} \times \textit{join}], \textit{value} \rangle\ , \emptyset \rangle \\
m_7 &= \langle\ \langle \text{source}, \text{location} \rangle\ ,\ \langle [\text{Feature\_Source} \times \textit{join}], [\text{has\_position} \times \textit{branch}], [\text{Location} \times \textit{branch}] \rangle\ , \emptyset \rangle \\
m_9 &= \langle\ \langle \text{location}, \textit{value}\{\text{``\^{}[\^{}.]+''}\} \rangle\ ,\ \langle [\text{Location} \times \textit{join}], [\text{start} \times \textit{branch}], \textit{value} \rangle\ , \emptyset \rangle \\
m_{10} &= \langle\ \langle \text{location}, \textit{value}\{\text{``[\^{}.]+''}\} \rangle\ ,\ \langle \text{Location} \times \textit{join}], [\text{end} \times \textit{branch}], \textit{value} \rangle\ , \emptyset \rangle \\
m_{12} &= \langle\ \langle \text{source}, [\text{qualifiers} \times \{\text{qualifiers}, \text{qualifiers/*/@name}\textit{value} = \text{``isolate''}\}] \rangle\ , \\
&\quad \langle [\text{Feature\_Source} \times \textit{join}], [\text{isolate} \times \textit{branch}] \rangle\ ,\ (m_{13}) \rangle \\
m_{13} &= \langle\ \langle \text{qualifiers}, \textit{value} \rangle\ ,\ \langle [\text{isolate} \times \textit{join}], \textit{value} \rangle\ , \emptyset \rangle
\end{aligned}
$$

In Figure 6.4, each mapping identifier is highlighted so they can be easily matched to the mapping definition given above. Mapping m_1 is a simple association between the `DDBJXML/ACCESSION` elements and the `Sequence_Data_Record/accession_id` elements. Mappings m_7 and m_8 contain simple regular expressions to assign the start and end locations contained in one string to different elements in the destination document. Mapping m_{12} contains a simple predicate expression that ensures the `qualifiers` element is transformed into an `isolate` element only when the string content of the `name` attribute is equal to `"isolate"` .

6.3 FXML-T Function Overview

The FXML-T library provides functions to convert XML schemas to `fxml:schema` structures, XML documents to `fxml:td` (typed documents), and M-Bindings expressed in XML to `fxml:binding` format.

$$\begin{aligned}
\texttt{xmls->fxml:schema} &: \langle string\rangle \rightarrow \langle fxml:schema\rangle\\
\texttt{xml->fxml:td} &: \langle string\rangle\langle fxml:schema\rangle \rightarrow \langle fxml:schema\rangle\\
\texttt{xml->fxml:binding} &: \langle string\rangle\langle fxml:schema\rangle\langle fxml:schema\rangle \rightarrow \langle fxml:binding\rangle
\end{aligned}$$

The `xmls->fxml:schema` function converts an XML schema document to an `fxml:schema`. It takes one string as input which refers to the location of the XML schema file. Files may be loaded from the local file system or over a network via HTTP. The `xml->fxml:td` function converts an XML document to an `fxml:td`. To perform this translation, an `fxml:schema` must be provided along with a reference to the location of the XML document. The `xml->fxml:binding` function converts an M-Binding document (a binding specified in XML) to an `fxml:binding` by taking a file location, and the input and output schema files. To transform an XML document, the `fxml:transform` function is used:

$$\begin{aligned}
\texttt{fxml:transform} &: \langle fxml:schema\rangle\langle fxml:td\rangle\langle fxml:schema\rangle\langle fxml:binding\rangle\\
&\rightarrow \langle fxml:td\rangle
\end{aligned}$$

Scheme S-Expression

```
(define binding
'("m1"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "DDBJ") (true))
   ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "ACCESSION") (true)))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Sequence_Data_Record") (join))
   ("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "accesion_id") (branch)))
  ;no local mappings
  ())
 ("m2"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "ACCESSION") (true))
   (value))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "accession_id") (join))
   (value))
  ;no local mappings
  ())
 ("m7"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "cds") (true))
   ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "location") (true)))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Feature_CDS") (join))
   ("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Location") (branch)))
  ;no local mappings
  ())
 ("m9"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "location") (true))
   (value "^[^.]+" ))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Location") (join))
   ("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "start") (branch))
   (value))
  ;no local mappings
  ())
 ("m10"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "location") (true))
   (value "[^.]+" ))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Location") (join))
   ("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "end") (branch))
   (value))
  ;no local mappings
  ())
 ("m12"
  global
  ;source mapping path
  (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "source") (true))
   ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers")
    ;predicate to ensure the qualifier value is organism
    (= (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers"))
        ("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers") (type . "*") (attribute . "name")))
        "isolate")))
  ;destination mapping path
  (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "Feature_Source") (join))
   ("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "isolate") (branch)))
  ; one local mapping
  ("m13"
   local
   ;source mapping path
   (("http://jaco.ecs.soton.ac.uk/schema/DDBJ" (element . "qualifiers") (true))
    (value))
   ;destination mapping path
   (("http://jaco.ecs.soton.ac.uk/ont/sequencedata" (element . "isolate") (join))
    (value))
   ;no local mappings
   ())))
```

Figure 6.4: Representation of Bindings in fxml-T

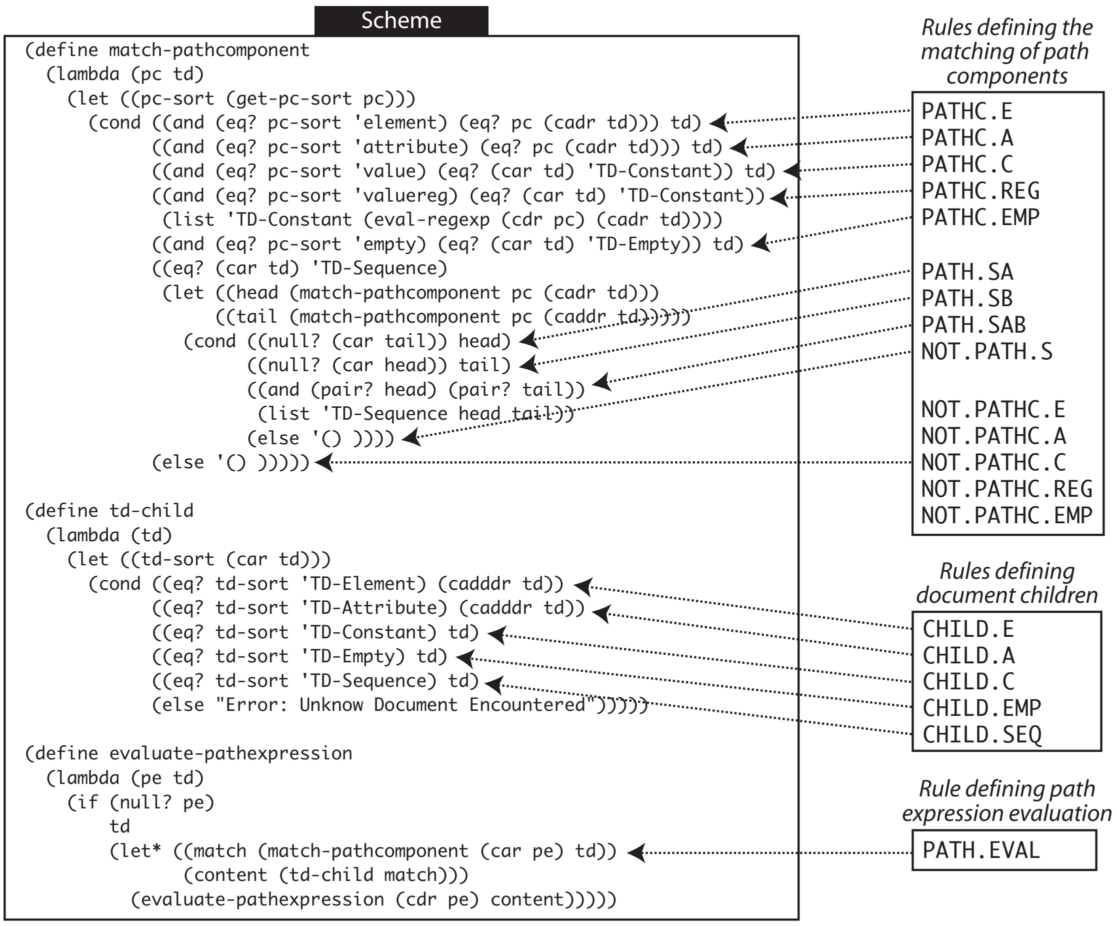
The `fxml:transform` function takes four arguments: an `fxml:schema` describing the source document, a source `fxml:td`, an `fxml:schema` describing the destination document, and an `fxml:binding`. The output is an `fxml:td` which is the result of the application of mappings in the `fxml:binding` to the source document according to the rules specified in Chapter 5, Section 5.5.3. Finally, we provide the `fxml:td->xml` function which allows us to convert an `fxml:td` to conventional XML syntax.

$$\texttt{fxml:td->xml} \;:\; \langle fxml:td\rangle \rightarrow \langle string\rangle$$

6.3.1 Transformation Rules in FXML-T

FXML-T is implemented in SCHEME because its inherent data structures (s-expressions) correlate closely to the structures used in FXML-M. When defining code to perform document translations, it is simple to implement the rules defined earlier in Chapter 5. To highlight the correspondence between FXML-M rules and the SCHEME functions that implement them, we give SCHEME code to evaluate document paths in Figure 6.5 and the rule names that they implement. The function `match-pathcomponent` takes a path component (`pc`) and a typed document (`td`) as input. This function implements the rules defined earlier in Chapter 5, Figure 5.4 for the matching of path components. The function `get-pc-sort` returns a symbol denoting the kind of component referenced which will be one of `element, attribute, value, valuereg, or empty`. Once the path component sort has been determined, simple conditional cases check that the path component referenced matches the typed document passed as input. If no rules for matching are true, then the empty list is returned.

The `td-child` function takes a typed document (`td`) as input and returns the child document, as defined by the rules in Chapter 5, Figure 5.5. Finally, the `evaluate-pathexpression` function is shown which implements the rule for evaluating path expressions (Figure 5.6 in Chapter 5). This function recurses through the sequence of path components, matching them against each typed document `td` and returning the contents. Because of this clear relationship between SCHEME



```
Scheme
(define match-pathcomponent
  (lambda (pc td)
    (let ((pc-sort (get-pc-sort pc)))
      (cond ((and (eq? pc-sort 'element) (eq? pc (cadr td))) td)
            ((and (eq? pc-sort 'attribute) (eq? pc (cadr td))) td)
            ((and (eq? pc-sort 'value) (eq? (car td) 'TD-Constant)) td)
            ((and (eq? pc-sort 'valuereg) (eq? (car td) 'TD-Constant))
             (list 'TD-Constant (eval-regexp (cdr pc) (cadr td))))
            ((and (eq? pc-sort 'empty) (eq? (car td) 'TD-Empty)) td)
            ((eq? (car td) 'TD-Sequence)
             (let ((head (match-pathcomponent pc (cadr td)))
                   ((tail (match-pathcomponent pc (caddr td)))))
               (cond ((null? (car tail)) head)
                     ((null? (car head)) tail)
                     ((and (pair? head) (pair? tail))
                      (list 'TD-Sequence head tail))
                     (else '() ))))
            (else '() )))))

(define td-child
  (lambda (td)
    (let ((td-sort (car td)))
      (cond ((eq? td-sort 'TD-Element) (cadddr td))
            ((eq? td-sort 'TD-Attribute) (cadddr td))
            ((eq? td-sort 'TD-Constant) td)
            ((eq? td-sort 'TD-Empty) td)
            ((eq? td-sort 'TD-Sequence) td)
            (else "Error: Unknow Document Encountered")))))

(define evaluate-pathexpression
  (lambda (pe td)
    (if (null? pe)
        td
        (let* ((match (match-pathcomponent (car pe) td))
               (content (td-child match)))
          (evaluate-pathexpression (cdr pe) content)))))
```

FIGURE 6.5: The correspondence between FXML-M transformation rules and the SCHEME code for FXML-T

function definitions and FXML-M rules, extensions or changes to the formalism can be easily implemented in FXML-T.

6.3.2 Transformation Algorithm and Complexity Analysis

To derive the complexity of the FXML-T transformation algorithm, we break down the translation process into a number of small functions that can be analysed individually. Pseudocode is used to present the transformation algorithm, and is given in listings 6.1 and 6.2. In FXML-T, components can be dereferenced by their namespace and local name using a component hash table. The use of hash tables ensures component dereferencing occurs in constant time, providing the hash table is sized appropriately. When reading an XML schema using the FXML-T library, a heuristic is used to size the component hash table based on the file size of the XML schema. Like the component hash table, mappings are indexed in the binding hash table by their source mapping path's first component so that applicable mappings for any given document can found in constant time.

When reading the pseudocode, parts of a structure are referenced using a `structure.part` notation. For example, a mapping contains a source mapping path, a destination mapping path, and local binding. The source mapping path is denoted by `mapping.sourcemappingpath`, the destination mapping path is denoted by `mapping.destinationmappingpath`, etc. Since these parts of a structure can be obtained directly, they are considered to operate in constant time, or $O(1)$. A number of basic functions are also assumed: `head(x)` and `tail(x)` are used in cases when a structure, such as a mapping path, is a list and either the first element or the rest of the list are required. We describe the pseudocode functions listed below:

- `transform(td, bindingmap)` - (line 1)
  The `transform` function consumes a source typed document and a binding hash table as input. The set of compatible mappings is retrieved from the binding hash table (line 2) and evaluated against the source typed document (line 3) to give a set of destination creation pairs. These pairs are then used to construct the destination document (line 4) which is returned as the function output.

- `evaluate(compatiblemappings, td, bindingmap)` - (line 8)
  The `evaluate` function takes a set of compatible mappings, a source document, and the binding hash table. Each mapping in the set of compatible mappings is evaluated separately (line 11) to generate a destination creation pair that is added to a result set.

- `evaluatemapping(mapping, td, bindingmap)` - (line 15)
  A mapping contains a source mapping path, a destination mapping path, and a set of local bindings. The `evaluate-mapping` function evaluates the source mapping path against the source document (line 20), obtaining a matched document as output. This matched document is itself translated (line 21) using the transform function defined above, with local bindings added to the binding hash table.

- `evaluate-smpath(smpath, td)` - (line 26)

  The `evaluate-smpath` function takes the first source mapping pair referenced in the source mapping path and extracts the path component (line 28) and predicate (line 29). The path component is matched against `td` (line 30) and the predicate is evaluated (line 31) to ensure the mapping is valid for application. If the predicate evaluates to true, the child document of the matched document is obtained (line 32). If the source mapping path contains more source mapping pairs, a recursion is made (line 34), passing the tail of the source mapping path and the child of the matched document as input. If there are no more pairs to process in the source mapping path, the child of the matched document is returned.

- `match-pathcomponent(pc, td)` - (line 41)

  To match path components to a typed document, the kind (or variety) of the path component is determined (line 42). The path component `pc` is then checked against `td` to see if they match (lines 44 - 53). If `td` is a sequence, a recursive call is made on each of the documents in the sequence (lines 55 and 56). Either both documents in the sequence match (line 57), only one is matched (lines 61 and 63), or neither match (line 59).

- `child-td(td)` - (line 69)

  The `child-td` function determines the kind of typed document passed as input and returns the appropriate content. If `td` is either an attribute or element, the tail is returned (where tail is the content document). If `td` is the empty document, a constant, or a sequence, `td` itself is returned.

- `predeval(predicate, td)` - (line 77)

  This function finds the kind of predicate passed as input (line 78) which will be one of: `exists`, `neg`, `and`, `or`, `less`, `greater`, `equal`, or `true`. If a predicate contains a `patom` (where a predicate atom is either a path expression or a constant), such as the `exists` predicate, `predicate.patom` is used to obtain the predicate atom. For the other cases, `predicate.subpredicate` is used to obtain sub-predicates that are used in the definition of a parent predicate, for example, the `and` predicate, that evaluates to true when both sub-predicates also evaluate to true.

- `apply(patom, td)` - (line 92)

  The application of a predicate atom to a typed document is simple: either the predicate atom is a path expression (line 94) that can be evaluated using the `evaluate-pe` function; or the predicate atom is a constant, in which case the constant itself is returned (line 95).

- `evaluate-pe(pe, td)` - (line 98)

  The `evaluate-pe` function consumes a path expression and a typed document. The first path component in the path expression is matched against `td` (line 99) and the child of the result is taken. If more path components are defined in the path expression, a recursive call is made, passing the tail of the path expression and the child of the matched document as input. If there are no more path components in the path expression, the child of the matched document is returned.

- `construct(dpairs)` - (line 107)

  The `construct` function takes a set of destination creation pairs as input and uses them to construct the destination document. The set of destination creation pairs is split into two subsets called `rjoin` and `rbranch`. Each of these is used separately to construct destination documents (lines 110 and 111) that are then combined in a sequence to give the destination document.

- `construct-rjoin(rjoin)` - (line 115)

  When constructing a destination document from `rjoin`, a new set of destination creation pairs is constructed by iterating through each pair in `rjoin` and removing the head of the destination mapping path (line 118). The first component referenced (`x`) in each pair's destination mapping path is determined (line 120) and its type (`t`) is obtained (line 121). The `construct` function is then called using the new set of destination creation pairs (line 122) to get a content document. The `construct-rjoin` function returns a new document created using the component `x`, of type `t`, with content `contentdocument`.

- `construct-rbranch(rbranch)` - (line 126)

  Each destination creation pair in `rbranch` is used to construct a separate document (line 129). All these documents are then combined using the `make-sequence` function to create the destination document (line 130).

- `construct-pair(pair)` - (line 133)
  This function consumes a destination creation pair (composed of a destination mapping path and a content document) and produces a destination document. If there is more than one destination mapping pair in `pair`'s destination mapping path (line 135), the first component referenced (`x`) is obtained (line 136), and its type (`t`) is determined (line 137). A new destination creation pair is then constructed (line 138) using the rest of the destination mapping path and the content document. This new destination creation pair is used to create a set of destination creation pairs with only one pair so it can be constructed using the `construct` function. If there is are no more destination mapping pairs in `pairs`'s destination mapping path (line 141), the first component referenced (`x`) is found (line 142). Based on the variety of `x` (i.e. `attribute`, `element`, `constant`, etc.), a destination document can be created with the contents from `pair.contentdocument`.

To calculate the complexity of the mapping evaluation algorithm presented above, we take a bottom-up approach, calculating the complexity of each function used, starting with the matching of path components. The `match-pathcomponent` function consumes a path component (`pc`) and a type document (`td`). If `td` is an attribute, element, constant, or the empty document, then the function returns in constant time $O(1)$. If `td` is a sequence of two typed documents then `match-pathcomponent` is called on each of them. The first of the two documents in the sequence (or the head) must be an element, attribute, constant, or the empty document, and the second (or tail) may be any kind of typed document (i.e. it could contain another sequence). Because of this linked-list structure, we can consider a typed document that is a sequence to be a list of typed documents with size n. Therefore, the worst case complexity of the `match-pathcomponent` function is $O(n)$, where n is the number of components contained in a sequence. For the rest of this analysis, we refer to the size of a typed document as n, where n is the number of elements, attributes, constants, or occurrences of the empty document.

Child documents are obtained using the `child-td` function. Since this function only checks the kind of the typed document passed as input and directly returns its content (when the kind is element or attribute), or itself (empty document, constant, or sequence), it operates in constant time — $O(1)$. The `evaluate-pe` function is used to evaluate a path expression (`pe`) against a typed document (`td`). Given that a path expression is a list of path components of size m, and each path component is matched against a typed document (itself of size n) in $O(n)$ time, the complexity of the `evaluate-pe` function is $O(m, n)$ where m is the number of components in the path expression and n is the size of the typed document.

Predicates are expressions that either contain predicate atoms (e.g. `exists`, `less`, `greater`, or `equal`), or other sub-predicates (e.g. `neg`, `and`, and `or`). We assume the size of a predicate (written p) is equal to the total number of predicate atoms in the expression, including those defined in sub-predicates. The `apply` function is used to apply a predicate atom to a typed document and executes in either $O(1)$ time (when the predicate atom is a constant), or in $O(m, n)$ time (when the predicate atom is a path expression of size m). Therefore, the complexity of the `predeval` function is $O(p, n)$, where n is the size of the typed document and p is the number of predicate atoms in the predicate expression.

To evaluate a source mapping path (function `evaluate-smpath`) with q pairs, the path component of each pair is matched against `td` (where `td` is the child of the evaluation of the previous pair in the source mapping path, or the source document for the first pair), and the predicate in each pair is matched against the result of `match-pathcomponent(pc,td)`. Therefore, the complexity of the `evaluate-smpath` is $O(q, n)$ where q is the size of the source mapping path and n is the size of the typed document.

To construct a destination document from a set of destination creation pairs, the `construct` function (line 107) is used. A set of destination creation pairs contains d pairs, each with r number of destination mapping pairs in their destination mapping path. The `construct` function splits the set of destination creations pairs into `rjoin` and `rbranch` and evaluates them separately using the `construct-rjoin` and `construct-rbranch` functions. To construct a destination document from `rjoin` (line 115), each destination creation pair has its first destination mapping pair removed from its destination mapping path (`tail(pair.dmpath)`). A call is then made to the `construct` function, using the new set of destination creation pairs. Therefore, the construction of destination documents from `rjoin` operates in $O(r)$, where r is the number of destination mapping pairs in the destination mapping path of each destination creation pair. To construct a destination document from `rbranch` (line 126), each pair in `rbranch` is constructed separately using the `construct-pair` function. Hence, construction from `rbranch` occurs in $O(d)$, where d is the number of destination creation pairs. When the two functions for the construction of `rjoin` and `rbranch` are combined in the `construct` function, the resulting complexity is $O(d, r)$.

With the complexity of source mapping path application and destination document construction in place, we can now derive the complexity of the transformation process. Each time a document is transformed using the `transform` function, a set of compatible mappings, of size c, is retrieved from the binding hash table. An iteration through each of these compatible mappings is made, evaluating each mapping individually to construct a destination creation pair. These destination creation pairs are then combined to make a set of destination creation pairs of size d. As we stated earlier, the construction of the destination document is $O(d, r)$. Therefore, the complexity of the `transform` function is $O(n, c)$, where n is the size of the source document, and c is the number of compatible mappings. Through evaluation of the FXML-T library later in Section 6.5, we confirm this result and show that increasing source document size only increases the transformation time linearly.

```
transform(td, bindingmap){
    compatiblemappings <— Hashtable.lookup(bindingmap, td.componentName)
    resultset <— evaluate(compatiblemappings, td, bindingmap);
    resultdocument <— construct(resultset)
    RETURN resultdocument
}

evaluate(compatiblemappings, td, bindingmap){
    resultset <— emptyset
    foreach mapping in compatiblemappings
        resultset.add(evaluatemapping(mapping, td, bindingmap))
    RETURN resultset
}

evaluatemapping(mapping, td, bindingmap){
    smpath <— mapping.sourcemappingpath
    dmpath <— mapping.destinationmappingpath
    localbinding <— mapping.localbinding

    matcheddocument <— evaluate—smpath(smpath, td)
    result <— transform(matcheddocument, bindingmap.add(localbinding))

    RETURN [result . destinationpath]
}

evaluate—smpath(smpath, td){
    firstpair <— head(smpath)
    pc <— firstpair.pathcomponent
    predicate <— firstpair.predicate
    matched—td <— match—pathcomponent(pc, td)
    IF predeval(predicate, matched—td)
        child—td <— td—child(matched—td)
        IF tail(smpath)
            RETURN evaluate—smpath(tail(smpath), child—td)
        ELSE
            RETURN child—td
    ELSE
        RETURN empty
}

match—pathcomponent(pc, td){
    pc—kind <— kind(pc)
    CONDITIONAL
        pc—kind = element AND td = pc
            RETURN td
        pc—kind = attribute AND td = pc
            RETURN td
        pc—kind = value AND td = constant
            RETURN td
        pc—kind = valuereg AND td = constant
            RETURN eval—regexp(pc, td)
        pc—kind = empty AND td = empty
            RETURN td
        td = sequence
            head <— match—pathcomponent(pc, head(td))
            tail <— match—pathcomponent(pc, tail(td))
            IF head != null AND tail != null
                RETURN make—sequence(head, tail)
            IF head = null AND tail = null
                RETURN null
            IF head = null
                RETURN tail
            IF tail = null
                RETURN tail
    ELSE
        RETURN null
}

child—td(td){
    td—kind = kind(td)
    CASE td—kind OF
        attribute:  RETURN tail(td)
        element:    RETURN tail(td)
        empty:      RETURN td
        constant:   RETURN td
        sequence:   RETURN td
}
```

Listing 6.1: Pseudocode for the transformation algorithm

```
predeval(predicate, td){
    CASE predicate.kind OF
        exists:     result <– apply(predicate.patom, td)
                    IF result != empty RETURN true ELSE RETURN false
        neg:        RETURN ! predeval(predicate.subpredicate, td)
        and:        RETURN predeval(head(predicate.subpredicates), td) AND
                           predeval(tail(predicate.subpredicates), td)
        or:         RETURN predeval(head(predicate.subpredicates), td) OR
                           predeval(tail(predicate.subpredicates), td)
        less:       RETURN apply(head(predicate.patom) < apply(tail(predicate.patom))
        greater:    RETURN apply(head(predicate.patom) > apply(tail(predicate.patom))
        equal:      RETURN apply(head(predicate.patom) = apply(tail(predicate.patom))
        true:       RETURN true
}

apply(patom, td){
    CASE patom OF
        pathexpression:  RETURN evaluate–pe(patom.pe, td)
        constant:       RETURN patom.constant
}

evaluate–pe(pe, td){
    matched–td <– match–pathcomponent(head(pe), td)
    child–td <– td–child(matched–td)
    IF tail(pe)
        RETURN evaluate–pe(tail(pe), child–td)
    ELSE
        RETURN child–td
}

construct(dpairs){
    rjoin <– rjoin(dpairs)
    rbranch <– rbranch(dpairs)
    td–j <– construct–rjoin(rjoin)
    td–b <– construct–rbranch(rbranch)
    RETURN make–sequence(td–j, td–b)
}

construct–rjoin(rjoin){
    dpairs <– emptyset
    for each pair in rjoin
        dpairs.add([tail(pair.dmpath) . pair.document]))

    x <– root(rjoin)
    t <– x.type
    content <– construct(dpairs)
    RETURN newdocument(x, t , content)
}

construct–rbranch(rbranch){
    resultdocument <– emptydocument
    for each pair in rbranch
        resultdocument.add(construct–pair(pair))
    RETURN make–sequence(resultdocument)
}

construct–pair(pair){
    dmpath <– pair.dmpath
    IF tail(dmpath)
        x <– head(dmpath).component
        t <– type(x)
        dpairs <– {[ tail(dmpath) . pair.document]}
        resultdocument <– construct(dpairs)
        RETURN newdocument(x, t, resultdocument)
    ELSE
        x <– head(dmpath).component
        CASE x.kind OF
            element:    RETURN newdocument(x, type(x), pair.contentdocument)
            attribute:  RETURN newdocument(x, type(x), pair.contentdocument)
            constant:   RETURN pair.contentdocument
            empty:      RETURN emptydocument
}
```

LISTING 6.2: Pseudocode for the transformation algorithm

6.4 The Configurable Mediator

The C-Mediator is a component that consumes *M*-Binding documents and uses them to direct the transformation of data from one format to another via an intermediate OWL representation. This process is broken into three stages: (i) conversion from the source XML format to OWL (conceptual realisation); (ii) modelling of the OWL concept instance; (iii) conversion from OWL to a destination XML format (conceptual serialisation). Stages (i) and (ii) are performed by the Translation Engine that is implemented using the FXML-T functions defined in Section 6.3. Figure 6.6 shows how these functions are combined to create the Transformation Engine.

The Transformation Engine takes four inputs: a source XML schema, a source XML document, a destination XML schema and an *M*-Binding in XML format. The `xmls->fxml:schema` function is used to convert the source and destination

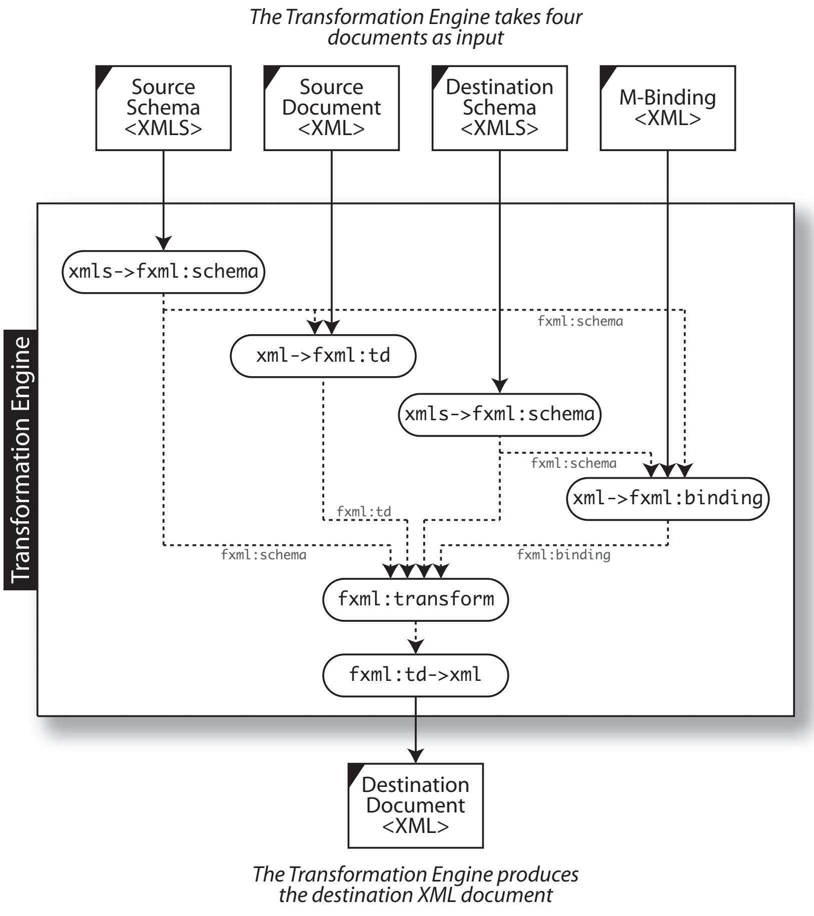


Figure 6.6: The Transformation Engine

XML schemas to `fxml:schema` structures. The source document is converted to an `fxml:td` using the `xml->fxml:td` function (consuming the source schema already converted to an `fxml:schema`). The *M*-Binding document is converted to an `fxml:binding` and then passed with the source `fxml:td`, source `fxml:schema`, and destination `fxml:schema` to the `fxml:transform` function. Once the document translation has been completed, the output is converted from an `fxml:td` to an XML document using the `fxml:td->xml` function.

After the initial conversion from the source XML format to an OWL concept instance (serialised in XML), the concept instance must be validated against its ontology definition. The C-MEDIATOR uses JENA to perform this stage of the mediation, creating an inference model from the ontology definition and importing the concept instance into it. During this stage, concept hierarchies are calculated and any instances imported are classified. From the perspective of our use case, this means that the output from the DDBJXML service (a *DDBJ_Sequence_Data_Record* concept) is also classified as an instance of the *Sequence_Data_Record* concept. Therefore, input to a service consuming a *Sequence_Data_Record*, such as the NCBI-Blast service, is valid. The C-MEDIATOR and its interaction with our DWSI and the two target Web Services from our use case is illustrated in Figure 6.7. In this diagram, the C-MEDIATOR is shown converting data from DDBJXML format to FASTA format via an instance of the *Sequence_Data_Record* concept. We show all the documents necessary for each conversion process (e.g. XML schemas and *M*-Binding documents) and where they originate (e.g. WSDL definitions, manually specified or automatically generated). To illustrate the mechanics of the C-MEDIATOR, we follow the conversion process in four stages, as they are labelled in Figure 6.7:

1. The Dynamic WSDL Invoker (DWSI) consumes the `accession_id` and invokes the DDBJ service to retrieve a complete sequence data record. The document returned is of type DDBJXML.

2. The DDBJXML sequence data record is converted to an instance of the `sequence_data_record` concept using the Translation Engine. The Translation Engine consumes the sequence data record, the XML schema describing it (taken from the DDBJ WSDL definition), a schema describing a valid instance of the `sequence_data_record` concept (generated automatically by

the OWL-$\mathcal{XIS}$ generator), and the realisation M-Binding document. The Translation Engine produces an instance of the `sequence_data_record` concept which is imported into the Mediation Knowledge Base (a JENA store).

3. To transform the `sequence_data_record` concept instance to FASTA format, the Translation Engine is used again, this time consuming the OWL concept instance (in XML format), the schema describing it (generated by the OWL-$\mathcal{XIS}$ generator), the schema describing the output format (from the NCBI-Blast WSDL) and the serialisation M-Binding. The output produced is the sequence data in FASTA format.

4. The DWSI consumes the FASTA formatted sequence data record and uses it as input to the NCBI-Blast service.

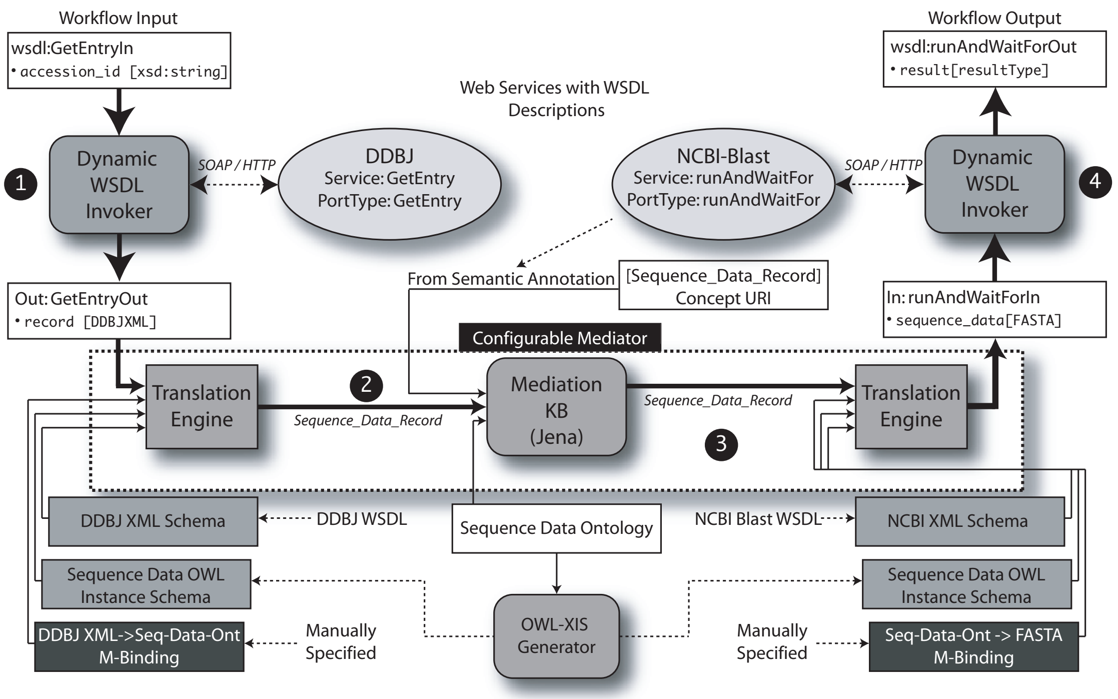


FIGURE 6.7: A detailed view of the Configurable Mediator in the context of our use case.

6.5 Evaluation

To evaluate our implementation of FXML-M, as well as the scalability of the language design itself, we devised four tests to examine the performance of our Translation Engine against increasing document sizes, increasing schema sizes, increasingly complex M-Binding composition, and a large set of Sequence Data Records. All tests were carried out using a 2.6Ghz Pentium4 PC with 1GB RAM running Linux (kernel 2.6.15-20-386) using unix utility `time` to record program user times. FXML-T is implemented in SCHEME and run using the Guile Scheme Interpreter v1.6 [53]. Results are averaged over 30 runs so plotted values are statistically significant at a 95% confidence interval.

6.5.1 Scalability

We test the scalability of FXML-T in two ways: by increasing input document size (while maintaining uniform input XML schema size), and by increasing both input schema size and input document size. The test hypothesis follows:

H1. Expanding document and schema size will increase the translation cost linearly.

For comparison, FXML-T is tested against the following XML translation tools:

- XSLT: Using Perl and the XML::XSLT module - `http://xmlxslt.sourceforge.net/`.
- XSLT: Using JAVA (1.5.0) and Xalan (v2.7.0) - `http://xml.apache.org/xalan-j/`.
- XSLT: Using Python (v2.4) and the 4Suite Module (v0.4) - `http://4suite.org/`.
- SXML: A SCHEME implemention for XML parsing and conversion (v3.0) `http://okmij.org/ftp/Scheme/SXML.html`.

Since FXML-T is implemented using an interpreted language, and Perl is also interpreted, we would expect them to perform slowly in comparison to JAVA and Python XSLT which are compiled[2]. Figure 6.8 shows the time taken to transform

[2] Although Python is interpreted, the 4Suite library is statically linked to natively compiled code

a source document to a structurally identical destination document for increasing document sizes. The maximum document size tested is 1.2 MB, twice that of the Blast results obtained in our use case. From Figure 6.8 we see that FXML-T has a linear expansion in transformation time against increasing document size: the correlation coefficient ($r^2 = \sigma_{xy}/\sigma_x\sigma_y$) is 0.916 (3 decimal places) where 1 is a straight line and 0 is evenly scattered data. Both Python and JAVA implementations also scale linearly with better performance than FXML-T due to JAVA and Python using compiled code. Perl exhibits the worst performance in terms of time taken, but a linear expansion is still observed. These results are summarised in the table presented in Figure 6.9. To compare each implementation, we calculate the line of best fit using the equation $y = mx + b$. The coefficient m for each implementation is listed in the table to convey the growth in transformation time. The difference in growth for each implementation to FXML-T is also listed, and presented as a percentage to assist the reader in comparison. For example, Perl is 94.0% slower than FXML-T, but Java is 62.4% faster.

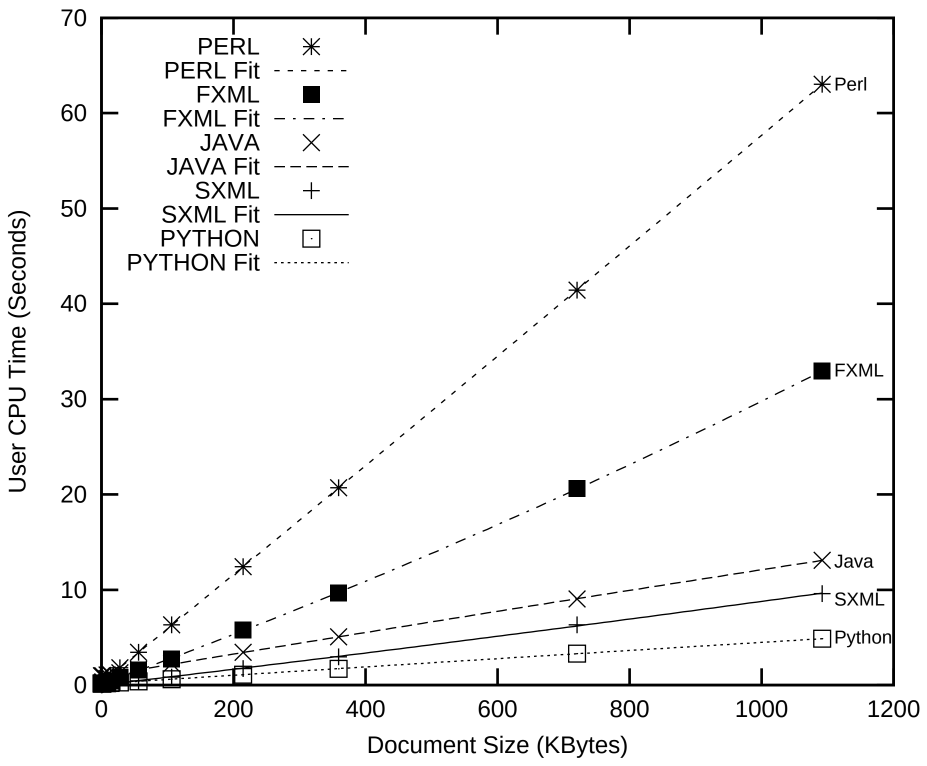


FIGURE 6.8: Transformation Performance against increasing XML document size

Doc Size (KB)	fxml (s)	perl (s)	java (s)	sxml (s)	python (s)
0.14	0.13	0.22	0.97	0.04	0.16
2.10	0.18	0.34	0.98	0.06	0.16
8.29	0.33	0.70	1.07	0.10	0.20
27.98	0.79	1.83	1.28	0.25	0.28
56.11	1.58	3.44	1.64	0.46	0.40
106.30	2.74	6.32	2.23	0.84	0.63
214.70	5.78	12.41	3.46	1.73	1.08
359.23	9.68	20.71	5.06	2.97	1.72
720.56	20.62	41.44	9.04	6.31	3.32
1091.65	32.94	63.05	13.10	9.61	4.86
m value	0.0296	0.0574	0.0111	0.0088	0.0043
	difference	0.0278	-0.0185	-0.0209	-0.0253
	percentage	94.0%	-62.4%	-70.4%	-85.4%

FIGURE 6.9: A summary of translation performance for increasing document sizes.

Our second performance test examines the translation cost with respect to increasing XML schema size. To perform this test, we generate structurally equivalent source and destination XML schemas and input XML documents which satisfy them. The XML input document size is directly proportional to schema size; with 2047 schema elements, the input document is 176KBytes, while using 4095 elements a source document is 378KBytes. Figure 6.10 shows translation time against the number of schema elements used.

Python and JAVA perform the best - a linear expansion with respect to schema size that remains very low in comparison to FXML-T and Perl. FXML-T itself has a quadratic expansion; however, upon further examination (see Figure 6.12), we find the quadratic expansion emanates from the XML parsing sub-routines used to read schemas and M-Bindings, whereas the translation itself has a cost linear to the size of its input (solid line in Figure 6.12). The SCHEME XML library used for XML parsing is common to FXML-T and SXML, hence the quadratic expansion for SXML also. Therefore, our translation cost would be linear if implemented with a suitable XML parser. A summary of these results is given in table format in Figure 6.11.

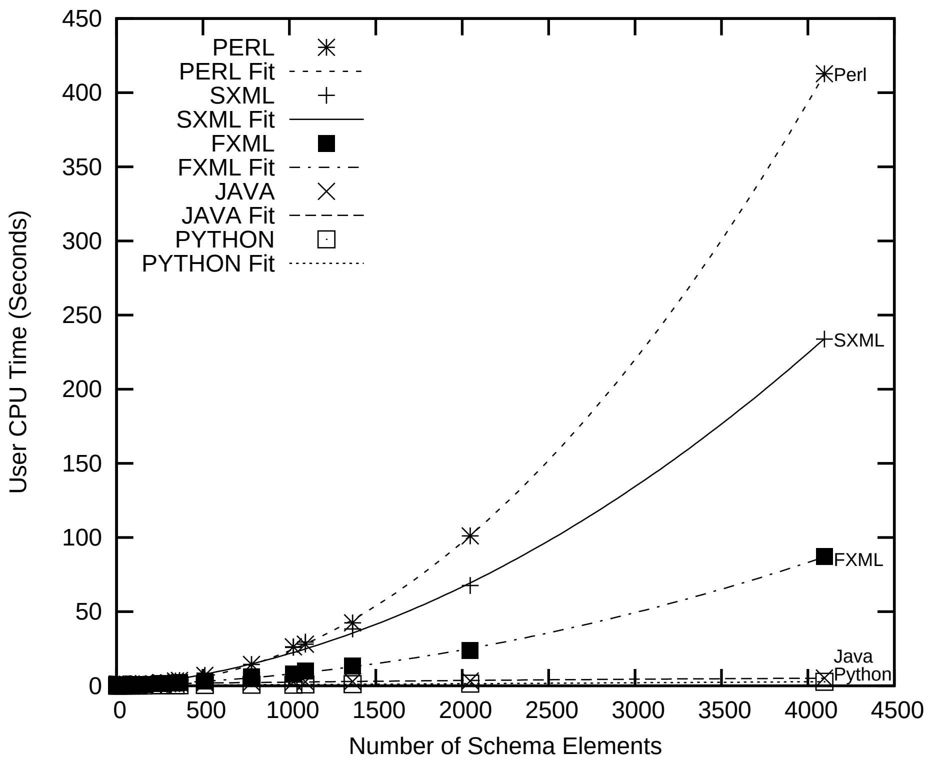


FIGURE 6.10: Transformation Performance against increasing XML schema size

Schema Size	fxml (s)	perl (s)	java (s)	sxml (s)	python (s)
3	0.14	0.22	0.04	1.03	0.17
15	0.19	0.24	0.07	1.08	0.18
85	0.49	0.51	0.21	1.23	0.21
156	0.88	0.99	0.38	1.45	0.26
255	1.42	2.08	0.76	1.60	0.31
511	3.29	7.04	3.69	1.98	0.46
1023	8.02	26.19	25.85	2.50	0.75
2047	23.69	101.10	67.61	3.32	1.42
4095	87.09	412.74	233.90	5.17	2.76
m value	0.0184	0.0850	0.0504	0.0011	0.0006
	difference	0.0666	0.0320	-0.0174	-0.0178
	percentage	361%	37%	-34%	-1693%

FIGURE 6.11: A summary of translation performance for increasing schema sizes.

6.5.2 Composition Cost

H2. Binding composition comes with virtually no performance cost.

One important feature of our translation language (FXML-M) is the ability to compose M-Bindings at runtime. This can be achieved by creating an M-Binding that

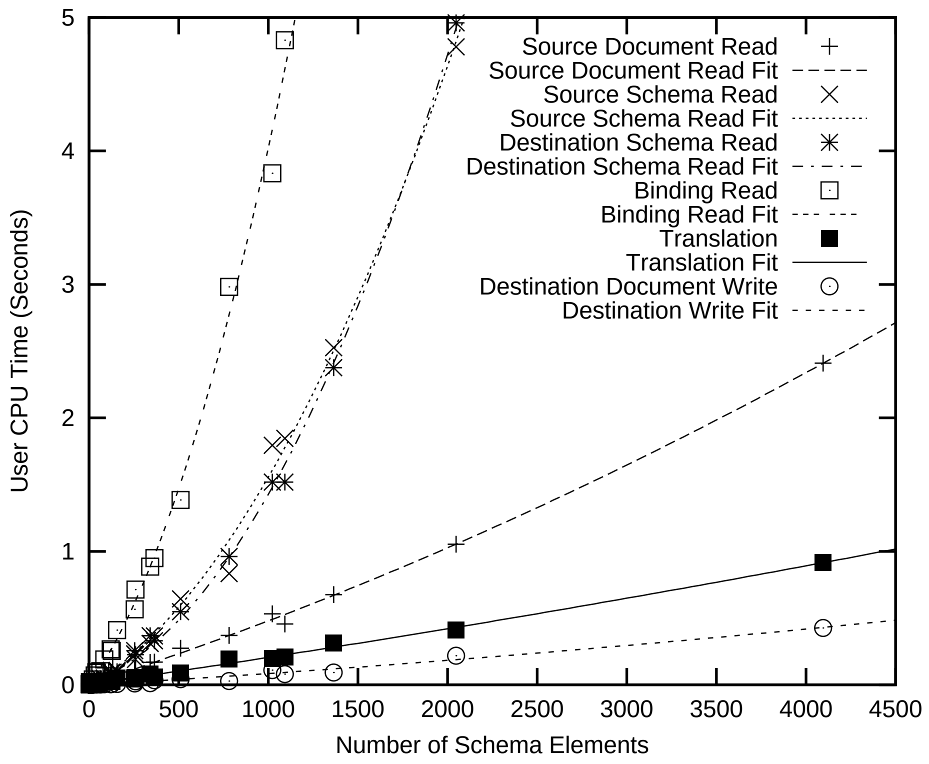


FIGURE 6.12: FXML-T transformation Performance breakdown against increasing XML schema size

includes individual mappings from an external M-Binding, or imports all mappings from an external M-Binding. For Service interfaces operating over multiple schemas, M-Bindings can be composed easily from existing specifications. Ideally, this composability should come with minimal cost. To examine M-Binding cost, we increased the number of M-Bindings imported and observed the time required to transform the document. To perform the translation, 10 mappings are required $m_1, m_2, \ldots, m_{10}$. M-Binding 1 contains all the required mapping statements: $B_1 = \{m_1, m_2, \ldots, m_{10}\}$. M-Binding 2 is a composition of two M-Bindings where $B_2 = \{m_1, \ldots, m_5\} \cup B_{2a}$ and $B_{2a} = \{m_6, \ldots, m_{10}\}$. To fully test the cost of composition, we increased the number of M-Bindings used and ran each test using 4 source documents with sizes 152Bytes, 411Bytes, 1085Bytes, and 2703Bytes. While we aim for zero composability cost, we would expect a small increase in translation time as more M-Bindings are included. By increasing source document size, a larger proportion of the translation time will be spent on reading in the document and translating it. Consequently, the relative cost of composing M-Bindings will be greater for smaller documents and therefore the increase in cost should be greater. Figure 6.13 shows the time taken to transform the same

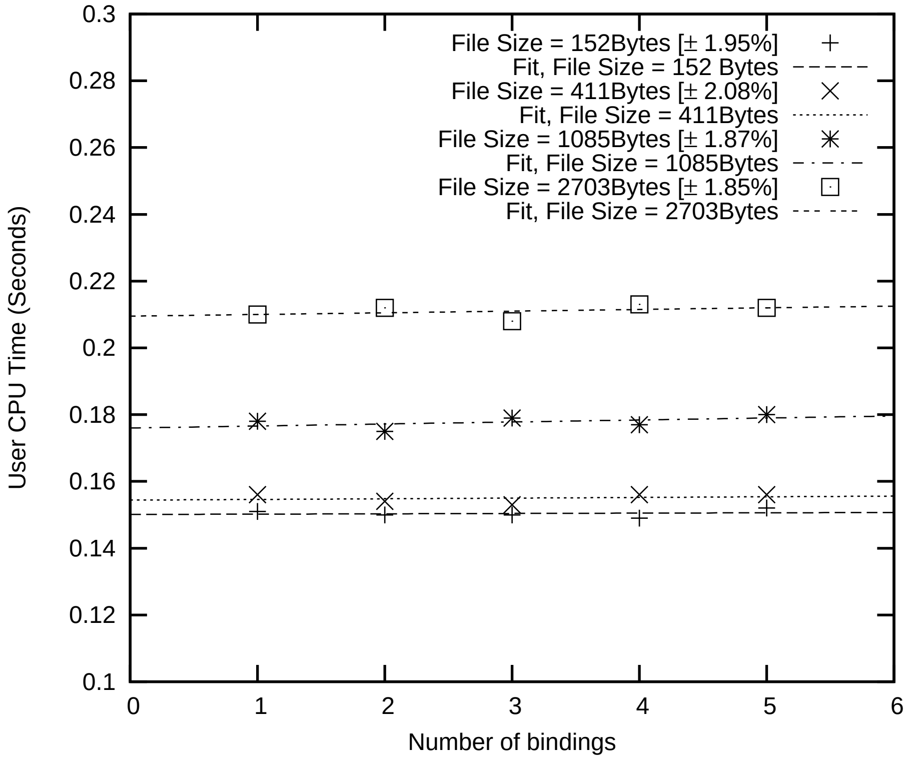


FIGURE 6.13: Transformation Performance against number of bindings

Number of Bindings	152KB	411KB	1085KB	2703KB
1	0.156	0.160	0.180	0.215
2	0.152	0.162	0.182	0.216
3	0.156	0.160	0.182	0.212
4	0.156	0.160	0.183	0.216
5	0.157	0.162	0.178	0.216
mvalue	0.0006	0.0002	-0.0003	0.0002
	max difference	0.005	0.026	0.059
	max percentage	103%	112%	118%

FIGURE 6.14: A summary of translation performance for increasing M-Binding composition.

four source documents against the same mappings distributed across an increasing number of M-Bindings. On the whole, a very subtle increase in performance cost is seen, and as expected, the increase is slightly larger for bigger documents. Again, a summary of values is given in Figure 6.14 where m values are shown to be very small. This indicates the that line of best fit is virtually flat, and therefore, the increase in translation cost is minute.

6.5.3 Bioinformatics Data Performance

H3. FXML-T *performs well in comparison to other transformation technologies when used to translate real bioinformatics data sets.*

To test the practicality of FXML-T, we randomly retrieve a large selection of sequence data records from the DDBJ-XML service and translate them to their corresponding OWL concept instance, serialised in XML. For comparison, the same translation is performed using an XSLT script with two different implementations: Perl and Python. Previous tests indicate that Perl XSLT performs worse than FXML-T and Python XSLT performs better, so we expect FXML-T values to fall roughly in the middle. Figure 6.15 is a plot of the time taken (in seconds) to transform a Sequence Data Record to an OWL concept instance against the size of the Sequence Data Record. On average, FXML-T translates a document in 60% of the time that Perl XSLT does, with an increase in translation time that is proportional to the size of the input document. Python performs much better, translating documents on average 50% quicker than FXML-T and with very little increase in translation time as document size increases. These results are also summarised in Figure 6.16.

6.5.4 Analysis

Hypothesis *H1* states that the performance cost of translation should be linear or better for FXML-T to be a scalable implementation. In Section 6.5.1, testing with increasing input document size shows FXML-T to have a linear increase in the cost of translation. Although a quadratic expansion is observed when schema sizes are increased, we discover that this performance overhead emanates from the XML parsing routines used and not the transformation cost which is shown to remain linear in Figure 6.12. Hypothesis *H2* states the M-Binding composition should ideally come with virtually zero performance cost. In Section 6.5.2, testing of increasingly complex M-Binding composition shows that the inclusion of mappings from other documents does not effect the translation performance in a significant way. Finally, to fulfil hypothesis *H3* and ensure FXML-T is a practical

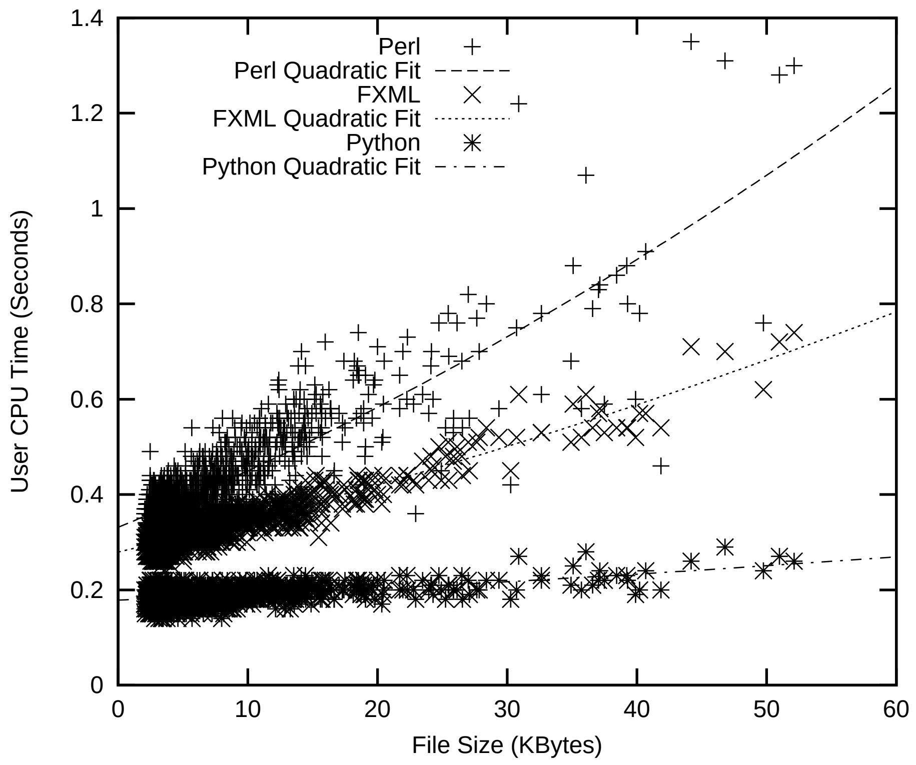


FIGURE 6.15: Transformation Performance against a random selection of Sequence Data Records from the DDBJ service

Doc Size (KB)	fxml (s)	perl (s)	python (s)
2.04	0.30	0.36	0.17
5.01	0.30	0.34	0.17
11.00	0.35	0.45	0.18
15.96	0.43	0.72	0.22
20.51	0.43	0.68	0.22
30.89	0.61	1.22	0.27
40.20	0.57	0.78	0.20
52.12	0.74	1.30	0.26
ihline mvalue	0.0088	0.0183	0.0017
	difference	0.0095	-0.0072
	percentage	107.6%	-81.2%

FIGURE 6.16: A summary of translation performance for bioinformatics data collected from DDBJ.

implementation, we test FXML-T against real bioinformatics data sources. Figure 6.15 illustrates that FXML-T performance is reasonable compared to other XSLT implementations.

6.6 Conclusions

FXML-T implements our transformation formalisation fully. It supports the translation of documents based on mappings between components within source and destination schemas. To express complex relations between elements, for example the mapping of elements based on other attribute values, FXML-T supports predicate evaluation. When the manipulation of string values is required, regular expressions may be used to extract characters of interest. FXML-T also provides an implementation of the core MSL constructs, namely schema components and typed documents. The MSL specification [26] does present inference rules that describe the process of document validation; i.e. the notation that an XML document conforms fully to the schema that describes it. However, we have not implemented this feature within FXML-T since third part schema validators can be used. Validation must be used otherwise it is possible to specify transformations that produce invalid documents.

Through evaluation of the FXML-T library against similar XML translation tools, we have shown that our implementation scales well when input document size is increased. While a quadratic expansion in translation time is observed when increasing schema sizes, we find that this increase emanates from the XML parsing subroutines used. The actual cost of translation remains linear with respect to input schema size. Therefore, our translation cost would be linear if more efficient XML parsing routines were used. In terms of M-Binding composition, our implementation performs very well: increasing the number of M-Bindings included has virtually no cost on the overall translation performance.

The languages FXML-M and XSLT [34] are obviously closely related because they both cater for XML translation. At a basic level, they provide operators that allow items of text in the source document to be replaced with different text in a destination document. However, FXML-M offers one significant benefit over XSLT:

FXML-M supports the composition of mappings in a predictable manner. With XML schema, the definition of elements, attributes and types can be imported from an external document. This is a useful feature when combining data from different sources because schema definitions do not need to be rewritten. If this were to occur with two XML schemas that both have an M-Binding to define the translation to another XML representation, the M-Binding definitions may also be imported, and therefore save considerable effort.

Chapter 7

Invocation and Discovery Architecture

The goal of the WS-MED architecture is to provide a generalised set of software components that can be exploited by any technology making use of Web Services standards (such as WSDL and XML schema) to translate XML data between different formats via an intermediate OWL representation. In the previous two Chapters, we have focused on the core syntactic mediation technology, namely the mapping language FXML-M (Chapter 5), its corresponding implementation FXML-T (Chapter 6), and the internal workings of the Configurable Mediator (C-MEDIATOR). While these contributions create the necessary infrastructure to support a scalable data translation approach, more software components are required to complete the big picture given in Figure 4.12 (Chapter 4). For example, analysis of current applications shows that the discovery and sharing of Type Adaptors is not well supported: most users create their own library of adaptors and rarely share them with other individuals.

If we are to consider the WS-MED architecture as a generic solution to the workflow harmonisation problem, we must support users and other software components in the sharing and discovery of Type Adaptors, the dynamic invocation of target services, and the generation of canonical XML representations for OWL concept instances. A more detailed list of these additional architecture requirements follows:

1. **Invocation of target services**
   In a constantly changing environment where services appear and disappear at any time, services may be discovered to achieve particular goals that have not been used before. Therefore, it is important for an invocation component to cater for the execution of previously unseen services defined using WSDL.

2. **Derive a canonical model from the intermediary representation**
   Since our intermediary-based mediation approach relies on a canonical XML representation of OWL concept instances to act as a *lingua franca*, a mechanism is required to automatically derive such a model. To be compatible with the C-MEDIATOR, an XML schema is required to validate OWL concept instances (called an OWL-$\mathcal{XIS}$).

3. **Discovery of Type Adaptors**
   To find autonomously the appropriate Type Adaptor to harmonise the flow of data between two services, a discovery and publishing facility is required that supports the advertising and retrieval of Type Adaptors based on their conversion capabilities. To conform to existing Web Service standards, we base this part of the architecture on a UDDI compliant registry.

In this Chapter, we present the WS-HARMONY architecture components that enable the execution of WSDL specified Web Services; the generation of OWL-$\mathcal{XIS}$ (OWL instance schemas); and the advertising, sharing, and discovery of Type Adaptors. We test Type Adaptor discovery cost in the context of workflow execution times and show that discovery time is minimal in comparison to the execution of target services. Our Dynamic Web Service Invoker is also tested against another Web Service invocation API (Apache Axis [10]) and is shown to be faster, particularly as the message size increases.

The contribution of this Chapter is a set of architecture components to satisfy the requirements above for automatic workflow harmonisation:

- A Dynamic Web Service Invoker that is able efficiently to execute previously unseen WSDL specified Web Services.
- An OWL XML instance schema (OWL-$\mathcal{XIS}$) generator that consumes OWL ontologies and produces XML schemas to validate concept instances.

- An approach for the description of Type Adaptor components using WSDL and a component to automatically generate WSDL definitions of M-Binding capability.
- A registration, sharing and discovery mechanism for Type Adaptors using the GRIMOIRES [93] service registry.

This Chapter is organised as follows: Section 7.1 gives an overview of WSDL, how services are typically invoked, why the invocation of previously unseen services is problematic, and how the Dynamic Web Service Invoker overcomes such problems. Section 7.2 discusses the relationship between OWL ontologies and XML schema, with an example to show the OWL-$\mathcal{XIS}$ generator at work using an algorithm that automatically creates OWL-$\mathcal{XIS}$. In Section 7.3, we concentrate on the description and discovery of Type Adaptors, presenting a uniform description method based on WSDL. Section 7.4 evaluates our Dynamic Web Service Invoker against a leading Web Service invocation API, and shows that the discovery of Type Adaptors in WS-HARMONY is insignificant compared to the execution of target services. Finally, we conlude in Section 7.5.

7.1 Dynamic Web Service Invocation

In this Section, we describe the problem faced by software components that are designed to enable the invocation of previously unseen Web Services. After a brief introduction to WSDL, and an explanation of the invocation problem, we present a solution that utilises a standardised XML view for service input and output messages. Finally, our implementation is presented in the form of the Dynamic Web Service Invoker (DWSI).

7.1.1 WSDL and Web Service Invocation

WSDL [33] is an XML grammar used to specify Web Services and how to access them. A WSDL document defines a service as a collection of endpoints, or ports.

Each port exposes a number of operations which the service supports. An operation is defined in terms of the input message it consumes and the output message it produces. A message has a number of uniquely named parts, the type of which is specified by a reference to an XML schema definition [41]. Custom schema definitions may also be included in the WSDL definition. The definition of the service, ports, operations and messages is done at an abstract level and bound to concrete execution models via the service bindings. The service binding specifies which type of protocol and datatype encoding is used for each operation, effectively stating how to invoke the service. By using a two tier model in which the service definition is given at an abstract level and its implementation is defined in terms of those abstractions, we are able to view many different Web Service implementations through a common interface. For example, a SOAP over HTTP binding, and a JMS [55] binding could both be specified for the same operation allowing clients from different platforms to utilise the same service.

We give an example WSDL document for the DDBJ-XML Bioinformatics service, used in our use case, in Listing 7.1. After the namespace declarations, the `<types>` element declares the types used by the service. The `<DDBJXML>` element definition is imported from an external schema. Following the type specification, the WSDL document declares two messages: the getEntryIn and getEntryOut messages, each of which has one part denoting the contents of the message. For the input message, there is only one part called `accession_id` of type `DDBJ:ACCESSION`. The output message also contains one part of type `DDBJ:DDBJXML` - a custom type to hold the Sequence Data Record. The `<portType name='DDBJPortType'>` element describes an endpoint which offers an operation called "GetEntry" which can be used to get Sequence Data Records. The input and output of this operation is specified by a reference to the previously defined WSDL messages. The `<binding name='DDBJBinding' type='tns:DDBJPortType'>` element provides a binding for the abstractly defined portType `GetEntry`.

A typical Web Service is implemented using SOAP [52] encoding over HTTP transport (as the DDBJ-XML service does). In this case, the service binding states that the message contents is placed inside a SOAP message and sent over HTTP. A SOAP message (or envelope) is an ordinary XML document that conforms to a specific schema defined at `http://www.w3.org/2001/12/soap-envelope`. The

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='DDBJService'
             targetNamespace='http://jaco.ecs.soton.ac.uk:8080/DDBJWrapper/ddbj'
             xmlns:tns='http://jaco.ecs.soton.ac.uk:8080/DDBJWrapper/ddbj'
             xmlns:DDBJ='http://jaco.ecs.soton.ac.uk/schema/DDBJ'
             xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
             xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
             xmlns:mime='http://schemas.xmlsoap.org/wsdl/mime/'
             xmlns:xsd='http://www.w3.org/2001/XMLSchema'
             xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
             xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
             xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <xsd:schema targetNamespace="http://jaco.ecs.soton.ac.uk:8080/DDBJWrapper/ddbj">
      <xsd:import namespace="http://jaco.ecs.soton.ac.uk/schema/DDBJ"
                 schemaLocation="http://jaco.ecs.soton.ac.uk/schema/DDBJ.xsd"/>
     </xsd:schema>
  </types>

  <message name='getEntryIn'>
    <part name='accession_id' element='DDBJ:ACCESSION'/>
  </message>
  <message name='getEntryOut'>
    <part name='record' element='DDBJ:DDBJXML'/>
  </message>

  <portType name='DDBJPortType'>
    <operation name='GetEntry'>
      <input name='getEntryIn' message='tns:getEntryIn'/>
      <output name='getEntryOut' message='tns:getEntryOut'/>
    </operation>
  </portType>

  <binding name='DDBJBinding' type='tns:DDBJPortType'>
    <soap:binding style='document' transport='http://schemas.xmlsoap.org/soap/http'/>
    <operation name='GetEntry'>
      <soap:operation soapAction='GetEntry'/>
      <input name='getEntryIn'>
        <soap:body use='literal'/>
      </input>
      <output name='getEntryOut'>
        <soap:body use='literal'/>
      </output>
    </operation>
  </binding>

  <service name='DDBJService'>
    <port name='DDBJPort' binding='tns:DDBJBinding'>
      <soap:address location='http://jaco.ecs.soton.ac.uk:8080/DDBJWrapper/ddbj'/>
    </port>
  </service>
</definitions>
```

Listing 7.1: WSDL Document describing the DDBJ-XML sequence retrieval service

```
<soap:envelope>
    <soap:body>
        <getEntryIn>
            <DDBJ:ACCESSION>AB000059</DDBJ:ACCESSION>
        </getEntryIn >
    </soap:body>
</soap:envelope>
```

Listing 7.2: Example SOAP envelope using RPC style

```
<soap:envelope>
    <soap:body>
        <DDBJ:ACCESSION>AB000059</DDBJ:ACCESSION>
    </soap:body>
</soap:envelope>
```

Listing 7.3: Example SOAP envelope using Document style

fundamental purpose of SOAP is to provide a standardised protocol for the exchange of information between distributed system components. In terms of Web Services, these components are the client and the service provider. During the invocation of a SOAP encoded WSDL Web Service, a SOAP envelope is created and the message parts are placed inside it. The format of this SOAP envelope depends on the binding style specified in the WSDL binding. The two types of style supported are document and rpc. With rpc style (see Listing 7.2), the child element of the `<soap:body>` node in the soap envelope has the same name as the WSDL operation name. The children of this operation node correspond to each of the message parts: each child element taking the same name as the message part name. With document style (see Listing 7.3), the children of the `<soap:body>` node correspond directly to the message parts (there is no node corresponding to the operation name - usually the `SOAPAction` HTTP header is used to distinguish different operations). They are not named according to the message part names, instead they are named after the XSD element they refer to.

In trivial cases the part type will be a simple, predefined type such as string or integer. However, XSD allows for the specification of complex types: an XML element that contains other elements (simple or complex). Existing Web Service APIs such as Apache or JAX-RPC use classes to encapsulate the XSD type and serialisers are registered to transform an object instance into the desired format. With complex types, such classes have to be created and compiled prior to execution and their corresponding serialisers registered at runtime. In terms of dynamic invocation, the problem is simple: without hard-coded classes to represent the complex types specified by the service provider, execution by the client is impossible. Therefore, we have developed an alternative message representation and invocation technique: instead of storing the message parts using classes, we define an XML representation of a WSDL message that is independent of binding style. This allows us to specify inputs to a WSDL service without knowing its implementation and hence provide dynamic invocation. It also simplifies the integration of the DWSI with the C-Mediator since all Web Services messages are instantiated in XML format. Our JAVA based Dynamic Web Service Invoker (DWSI) is able to invoke WSDL specified services when given inputs in this XML format, and returns the results in the same format. The following sections define the XML representation with examples and the interface to our DWSI.

7.1.2 XML representation of WSDL messages

The root element of the WSDL message XML representation takes the same name as the WSDL message name. Each of its child elements corresponds to a message part with each element taking the name of the message part as shown in Listing 7.4.

If complex types are used, they are represented as children of the `<part-name>` element. We show an example input message for invocation of the DDBJ-XML sequence retrieval service in Listing 7.5 and fragment of the output in Listing 7.6.

```
<message−name>
 <part−name> part contents </part−name>
 <part−name> part contents </part−name>
 ...
 <part−name> part contents </part−name>
</message−name>
```

Listing 7.4: An XML representation of a WSDL message.

```
<getEntryIn>
  <accession_id>
      <ACCESSION xmlns="http://jaco.ecs.soton.ac.uk/schema/DDBJ">AB000059</ACCESSION>
  </accession_id>
</getEntryIn>
```

Listing 7.5: Example Input Message for DDBJ-XML Sequence Retrieval Service

```
<getEntryOut>
  <record>
    <DDBJXML xmlns="http://jaco.ecs.soton.ac.uk/schema/DDBJ">
      <ACCESSION>AB000059</ACCESSION>
      ...
      <FEATURES>
        <source>
           <location>1..1755</location>
           <qualifiers name="isolate">Som1</qualifiers>
           <qualifiers name="lab_host">Felis domesticus</qualifiers>
           <qualifiers name="mol_type">genomic DNA</qualifiers>
           <qualifiers name="organism">Feline panleukopenia virus</qualifiers>
        </source>
        <cds>
          <location>1..1755</location>
          <qualifiers name="product">capsid protein 2</qualifiers>
          <qualifiers name="protein_id">BAA19020.1</qualifiers>
          <qualifiers name="translation">MSDGAV...</qualifiers>
        </cds>
      </FEATURES>
      <SEQUENCE>atgagtgatggagcagt..</SEQUENCE>
    </DDBJXML>
  </record>
</getEntryOut>
```

Listing 7.6: Example Output Message for DDBJ-XML Sequence Retrieval Service (fragment only).

Since no proper schema definition exists to describe this type of container, we have created a simple schema generator which will create the correct schema when passed a WSDL interface, port-type, service name, and desired operation. This software component is exposed as a Web Service (available at `http://jaco.ecs.soton.ac.uk:8080/schema`), where the WSDL location, port-type, service name, and desired operation are all passed as arguments. Listing 7.7 shows an example schema to describe the input and output messages of the "getEntry" operation provided by the DDB-XML service. Because our schema generator is exposed as a Web Service, the target namespace of the XML document can reference the schema generator (lines 4 - 8), and therefore, support the validation of WSDL input and output message documents. By using this XML representation of WSDL messages, we can view the data sent to and returned from any type of WSDL service execution through the same data representation.

7.1.3 Dynamic Web Service Invoker

Our Dynamic Web Service Invoker (DWSI) exposes one method named `invoke`, with the following signature:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tns="http://jaco.ecs.soton.ac.uk:8080/DDBJWrapper/ddbj"
            xmlns:DDBJ='http://jaco.ecs.soton.ac.uk/schema/DDBJ'
            targetNamespace="http://jaco.ecs.soton.ac.uk:8080/schema.jsp?
                             wsdllocation=http://jaco.ecs.soton.ac.uk/wsdl/ddbj.wsdl;
                             servicens=null;servicename=DDBJService;
                             porttypens=null;porttypename=DDBJPortType;
                             operation= GetEntry">
  <xsd:element name="GetEntryIn">
    <xsd:comlexType>
      <xsd:sequence>
        <xsd:element ref="accession_id" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:comlexType>
  </xsd:element>
  <xsd:element name="GetEntryOut">
    <xsd:comlexType>
      <xsd:sequence>
        <xsd:element ref="record" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:comlexType>
  </xsd:element>
  <xsd:element name="accession_id">
    <xsd:comlexType>
      <xsd:sequence>
        <xsd:element ref="DDBJ:ACCESSION" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:comlexType>
  </xsd:element>
  <xsd:element name="record">
    <xsd:comlexType>
      <xsd:sequence>
        <xsd:element ref="DDBJ:DDBJXML" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:comlexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.7: Example XML schema to wrap DDBJ-XML Sequence Retrieval Service input operation

```
public static Node invoke(String wsdlLocation, String operation,
                          String serviceNS, String serviceName,
                          String portTypeNS, String portTypeName,
                          Node inputDOM);
```

The parameters are:

- **wsdlLocation** - The URL of the WSDL document
- **operation** - The operation to call
- **serviceNS** - The service namespace
- **serviceName** - the service name
- **portTypeNS** - the port-type namespace

- **portTypeName** - the port-type name
- **inputDOM** - An org.w3c.dom.Node object holding the input XML
- **return** - An org.w3c.dom.Node object holding the output XML

Our current version supports SOAP over HTTP bindings only since they are the only ones used within our bioinformatics application. The DWSI supports both types of style (rpc and document) allowing it to invoke Web Services deployed on any platform including JAVA and .NET, a feature not adequately supported in any existing JAVA APIs. Evaluation of the DWSI is presented later in Section 7.4.1 where invocation of the DDBJ-XML service using the DWSI is compared to Apache Axis.

7.2 Generation of OWL Instance Schemas

We stated earlier in Chapter 4 that we simplify our transformation requirements for conceptual realisation and conceptual serialisation by assuming a canonical XML representation of OWL concept instances. This way, the realisation and serialisation translation process can be viewed as an XML to XML translation. While it is common for OWL users to specify OWL concepts and instances using an RDF/ XML syntax, XML schemas do not usually exist to validate them. Therefore, automated harmonisation can only be achieved if these schemas are generated. To present this idea, we use a simple vehicle ontology, illustrated in Figure 7.1. The *Vehicle* concept has two datatype properties (*number_of_wheels* and *number_of_seats*) and two subconcepts: *Van* and *Car*. Every vehicle has an *Engine* (which could be described by a more specific concept such as *Petrol* or *Diesel*) and a *Transmission*. Listing 7.8 shows the XML schema created by the OWL-$\mathcal{XIS}$ generator to validate instances from the vehicle ontology. The algorithm is outlined below with references to parts from the schema listing.

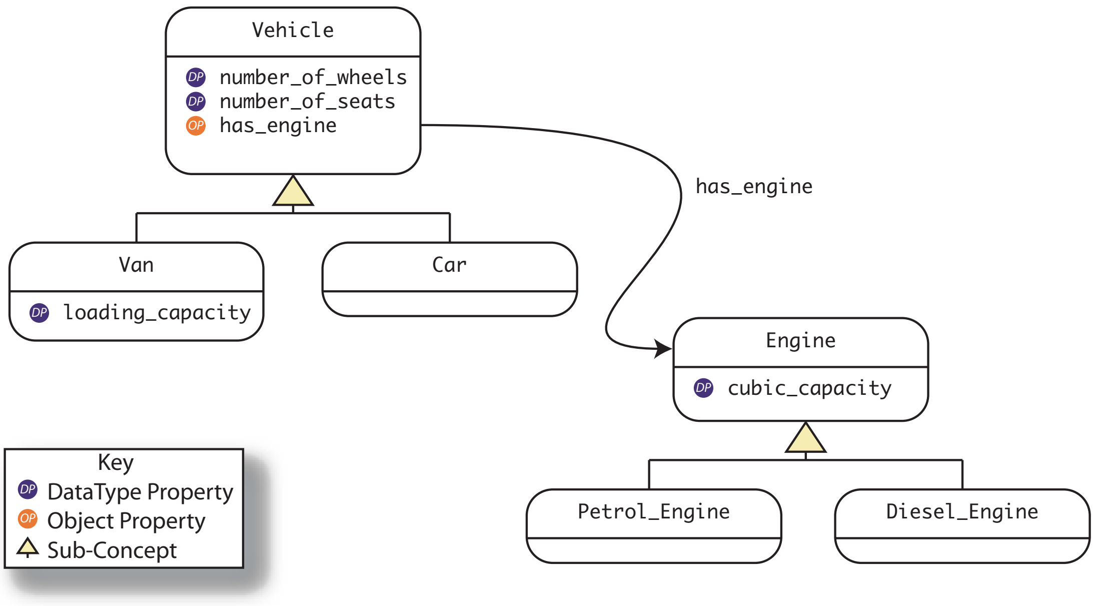


Figure 7.1: A simple vehicle ontology

7.2.1 Algorithm for XML Schema Generation

Klein *et al* [65] present an algorithm to generate XML schemas that validate OIL [56] ontology-containers. Using an adapted version of their algorithm to cater for OWL ontologies, we are able to generate XML schemas to validate OWL concept instances for a given ontology definition. The algorithm is outlined below:

1. **Materialise the hierarchy**

   OWL provides language constructs to specify concept hierarchies so a particular concept can be considered a more general classification than another. For example, the concept *Vehicle* can be considered more general than the concepts *Car* or *Van*. Subsumption, usually denoted as $C \sqsubseteq D$, is the reasoning processes through which the concept D (the *subsumer*) is checked to see if it is more general than the concept denoted by C (the *subsumee*). Reasoning engines, such as JENA, provide subsumption reasoning so when an ontology definition is loaded, all concept hierarchies are calculated automatically.

2. **Create an element for each concept**

   For every OWL concept in the ontology, an XSD element is created. For the vehicle ontology in Figure 7.1, the following elements would be created:

`<Vehicle>`, `<Van>`, `<Car>`, `<Engine>`, `<Petrol_Engine`, and `<Diesel_Engine>`. These can be found in lines 5 to 10 of Listing 7.8.

3. **Create an element for each property**
   For every OWL property in the ontology, an XSD element is created. For properties that link concepts to other concepts (called an **object property**), such as the *has_engine* property, the type of the element is a complex type. For properties that link concepts to literal values (called a **datatype property**), such as the *number_of_wheels* and *number_of_seats* properties, the type is the same as the type given in the OWL definition and is likely to be one of the predefined XSD types such as an integer or string. Property element definitions can be found in lines 13, 16-19 of Listing 7.8.

4. **Create a complex Type definition for each concept**
   Once the XSD elements have been created, an XML schema complex type is created for each concept. When creating the complex type, a list of all possible properties for that concept are extracted by checking the domain of all properties. The complex type is then specified as a sequence over these properties with any cardinality constraints from the property reflected using XML schema occurrence indicators. In cases where a concept is a subconcept of another, such as the *Car* concept in the vehicle ontology, XSD type extension is used to provide the inheritance of properties from the parent. See lines 22 - 63 of Listing 7.8 for complex Type definitions.

5. **Create a type definition for each property**
   Finally, a type definition is created for every property in the ontology. As we stated above, datatype properties are assigned a simple type and object properties are given a complex type. When object property types are created, the range of the property is examined and a list of possible concepts that property links to is determined. When an object property links to a concept which has sub concepts, such as the *has_engine* property in the vehicle ontology, the complex type is set to be a choice over any of the sub concepts, e.g. the *has_engine* complex type will be a choice of *Engine*, *Petrol_Engine*, or *Diesel_Engine*. The type definition for the *has_engine* property can be found in lines 66 - 72 of Listing 7.8.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://jaco.ecs.soton.ac.uk/ont/vehicle_ontology"
            targetNamespace="http://jaco.ecs.soton.ac.uk/ont/vehicle_ontology">
    <!-- Concept Elements -->
    <xsd:element name="Vehicle" type="Vehicle-TYPE"/>
    <xsd:element name="Van" type="Van-TYPE"/>
    <xsd:element name="Car" type="Car-TYPE"/>
    <xsd:element name="Engine" type="Engine-TYPE"/>
    <xsd:element name="Petrol_Engine" type="Petrol_Engine-TYPE"/>
    <xsd:element name="Diesel_Engine" type="Diesel_Engine-TYPE"/>

    <!-- Object Property Elements -->
    <xsd:element name="has_engine" type="has_engine-TYPE"/>

    <!-- Datatype Property Elements -->
    <xsd:element name="number_of_wheels" type="xsd:integer"/>
    <xsd:element name="number_of_seats" type="xsd:integer"/>
    <xsd:element name="loading_capacity" type="xsd:integer"/>
    <xsd:element name="cubic_capacity" type="xsd:integer"/>

    <!-- Concept Types -->
    <xsd:complexType name="Vehicle-TYPE">
        <xsd:sequence>
            <xsd:element ref="has_engine"/>
            <xsd:element ref="has_transmission"/>
            <xsd:element ref="number_of_wheels"/>
            <xsd:element ref="number_of_seats"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Van-TYPE">
        <xsd:complexContent>
            <xsd:extension base="Vehicle-TYPE">
                <xsd:sequence>
                    <xsd:element ref="loading_capacity"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Car-TYPE">
        <xsd:complexContent>
            <xsd:extension base="Vehicle-TYPE"/>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Engine-TYPE">
        <xsd:sequence>
            <xsd:element ref="cubic_capacity"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Petrol_Engine-TYPE">
        <xsd:complexContent>
            <xsd:extension base="Engine-TYPE"/>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Diesel_Engine-TYPE">
        <xsd:complexContent>
            <xsd:extension base="Engine-TYPE"/>
        </xsd:complexContent>
    </xsd:complexType>

    <!-- Property Types -->
    <xsd:complexType name="has_engine-TYPE">
        <xsd:choice>
            <xsd:element ref="Engine"/>
            <xsd:element ref="Petrol_Engine"/>
            <xsd:element ref="Diesel_Engine"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:schema>
```

Listing 7.8: Example XML Schema to validate instance of the vehicle ontology

When creating any elements or complex types, the namespace and local name of the concept is mirrored in the XML schema. This means that a uri pointing to a particular OWL concept or property also refers to the XML schema element that validates it. Pseudocode for the OWL instance schema generation algorithm is given in Listing 7.9. The `create-schema` function take three arguments: `concepts` (a materialised hierarchy of the concepts in the OWL ontology); `properties` (a hierarchy of the properties within the OWL ontology); and `schema` (the schema document to be created - initially emtpy). For every concept in the ontology, an element declaration is added to the schema using the concept namespace, the concept local name, and the name of the type (which is the local name concatenated with the string "-TYPE"). Once the element declaration has been added to the schema, a complex type definition is created using the `create-concept-complex-type`. This function iterates through all properties that have a domain equal to the concept passed in the argument, and creates a complex type that is equal to a sequence over these properties.

Once elements and types have been created for each concept, the `create-schema` function iterates through the list of properties and calls the `create-property` function. This function first checks the kind of property passed in the argument to ascertain whether it is an *object property* or a *datatype property*. If it is a datatype property, an element is created with the same XML type as the OWL property. If it is an object property, an element is created using the property's

```
create−schema(concepts, properties, schema){
  foreach concept in concepts
    schema.addelement(concept.ns, concept.localname, concept.localname + "−TYPE")
    create−concept−complex−type(concept, properties, schema)

  foreach property in properties
    create−property(property, concepts, properties, schema)
}

create−concept−complex−type(concept, properties){
  sequence <− empty
  foreach property in properties
    if property.domain = concept
      sequence.add(property)

  schema.addcomplexType(concept.ns, concept.localname + "−TYPE",
                        sequence, concept.parent)
}

create−property(property, concepts, properties, schema){
  if property.kind = datatype
    schema.addelement(property.ns, property.localname, property.type)
  else property.kind = object
    schema.addelement(property.ns, property.localname, property.localname + "−TYPE")
    sequence <− empty
    foreach concept_in_range in property.range()
      choice <− empty
      foreach subconcept in concept_in_range.subconcepts()
        choice.add(subconcept)

      sequence.add(choice)
    schema.addcomplexType(property.ns, property.localname + "−TYPE",
                          sequence, property.parent)
}
```

Listing 7.9: Pseudocode for the generation of OWL-$\mathcal{XIS}$

namespace and localname and a complex type is created. The complexType to define the element contents is a sequence over all the concepts that are in the range of the property. If any of these concepts has sub-concepts in the concept hierarchy, a choice indicator is used to specify that any of the sub-concepts are also valid.

The OWL-$\mathcal{XIS}$ generator is implemented using JAVA and the JENA toolkit. The OWL-$\mathcal{XIS}$ generator consumes an OWL ontology and produces an XML schema to validate instances of concepts from the given ontology and is exposed as a Web Service.

7.3 Type Adaptor Description and Discovery

To fully automate the workflow harmonisation process, it is necessary for the C-Mediator to be able to access the required resources (i.e. the serialisation and realisation M-Bindings) at runtime without user intervention. Given WSDL service interfaces that specify syntactic types (by references to XML Schema elements) and semantic service annotations that define semantic types (by reference to concepts within an OWL ontology), dataflow between services can be examined for inconsistencies. If the output syntactic type from a source service is different from the input type to a target service, they are not syntactically compatible. However, if the source output semantic type references the same concept (or is subsumed by the same concept) as the input to the target service, they are deemed semantically compatible. When this occurs, a query to registry can be made to find realisation and serialisation M-Bindings and a Type Adaptor can be created using the C-Mediator. In the following subsections, we explain how WSDL can be used to describe Type Adaptor capabilities and support the discovery of Type Adaptors through the use of a service registry.

7.3.1 Type Adaptor Discovery Requirements

There are many applications and tools that support the translation of data between different formats. XSLT [34] enables the specification of data translation in a script format using pattern matching and template statements. Such a script can be consumed by an XSLT engine to drive the translation of XML data to a different representation. Other forms of Type Adaptors are not so transparent; translation programs are often created using languages such as JAVA and Perl. In other cases, a Type Adaptor may take the form of a distinct mediator Web Service, described by WSDL and executed using SOAP over HTTP. When data flow within a workflow links two syntactically incompatible interfaces (i.e. the output type from the source service is different to the input type of the destination service), Type Adaptors must be inserted to harmonise differences in representation. As we stated in Chapter 2, this is currently a manual process that must be carried out at workflow design time.

In Chapter 4, Section 4.1, two mediation approaches were identified: direct and intermediary based. With a direct approach, one type adaptor is required to translate from a source format straight to a destination format. With an intermediary based approach, where data is transformed to common representation expressed using an ontology language, two type adaptors are required: one for conceptual realisation and one for conceptual serialisation, illustrated in Figure 7.2. In these scenarios, it is assumed that the necessary adaptor components are known and inserted into the workflow (in the case of direct mediation), or consumed by the Configurable Mediator (for intermediary based mediation).

Since current Grid and Web Services infrastructures provide no mechanism to describe, advertise or discover Type Adaptors, adaptor development is often ad hoc: users create translation components on demand, even though other users way have already engineered them. Individuals can build their own libraries of adaptors, but are unable to obtain those created by others without direct intervention, for example, by email or file transfer. To reduce user effort through the sharing of adaptor components, as well as supporting the retrieval of Type Adaptors for automated harmonisation, an advertising and discovery mechanism is required that enables users and programs to get Type Adaptors according to type conversion capabilities. We break down the requirements for such a system as follows:

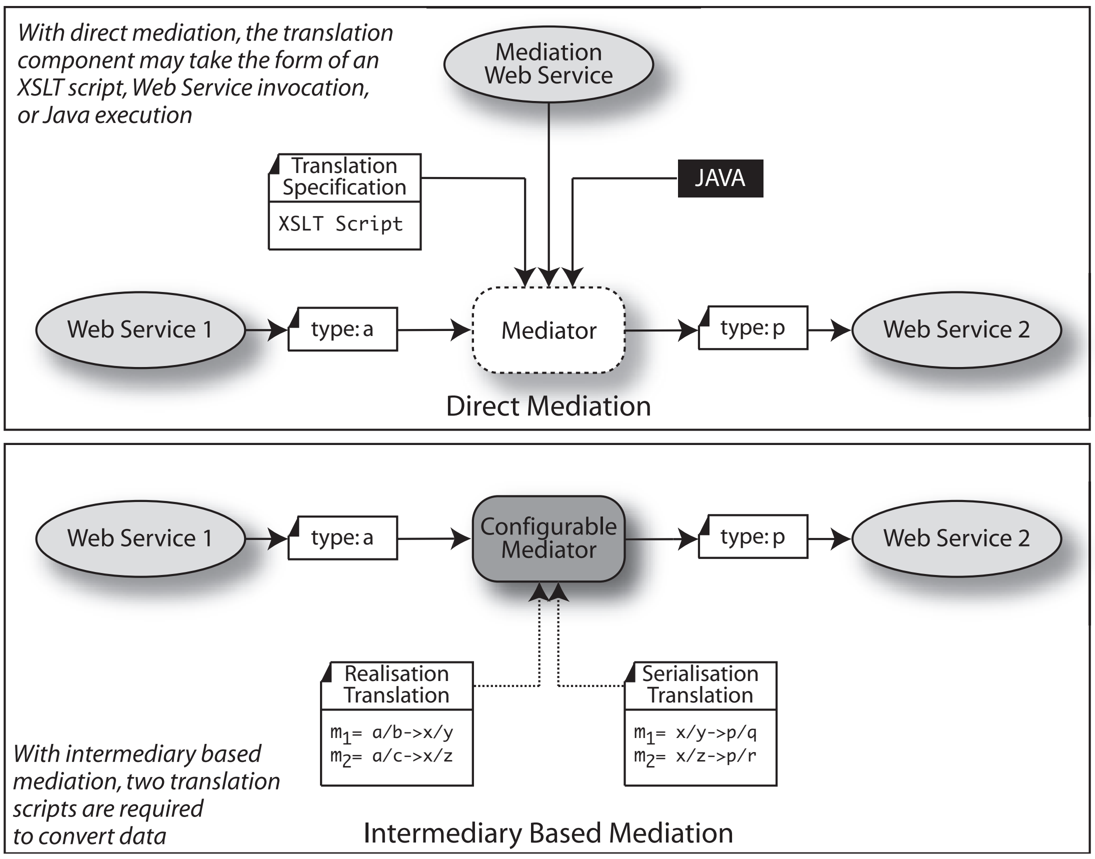

With direct mediation, the translation component may take the form of an XSLT script, Web Service invocation, or Java execution
Mediation Web Service
Translation Specification
XSLT Script
JAVA
Web Service 1
type: a
Mediator
type: p
Web Service 2
Direct Mediation
Web Service 1
type: a
Configurable Mediator
type: p
Web Service 2
Realisation Translation
m1= a/b->x/y
m2= a/c->x/z
Serialisation Translation
m1= x/y->p/q
m2= x/z->p/r
With intermediary based mediation, two translation scripts are required to convert data
Intermediary Based Mediation

FIGURE 7.2: Differences in execution for direct and intermediary based mediation

1. **A standard way to describe Type Adaptor capabilities**

   To support the discovery of adaptor components according to their functionality while remaining agnostic of their implementation, a description approach must be employed that specifies: (i) the abstract functionality of the adaptor in terms of the source type consumed and the destination type produced; (ii) the concrete execution model showing how to invoke the component.

2. **A repository to store Type Adaptor information**

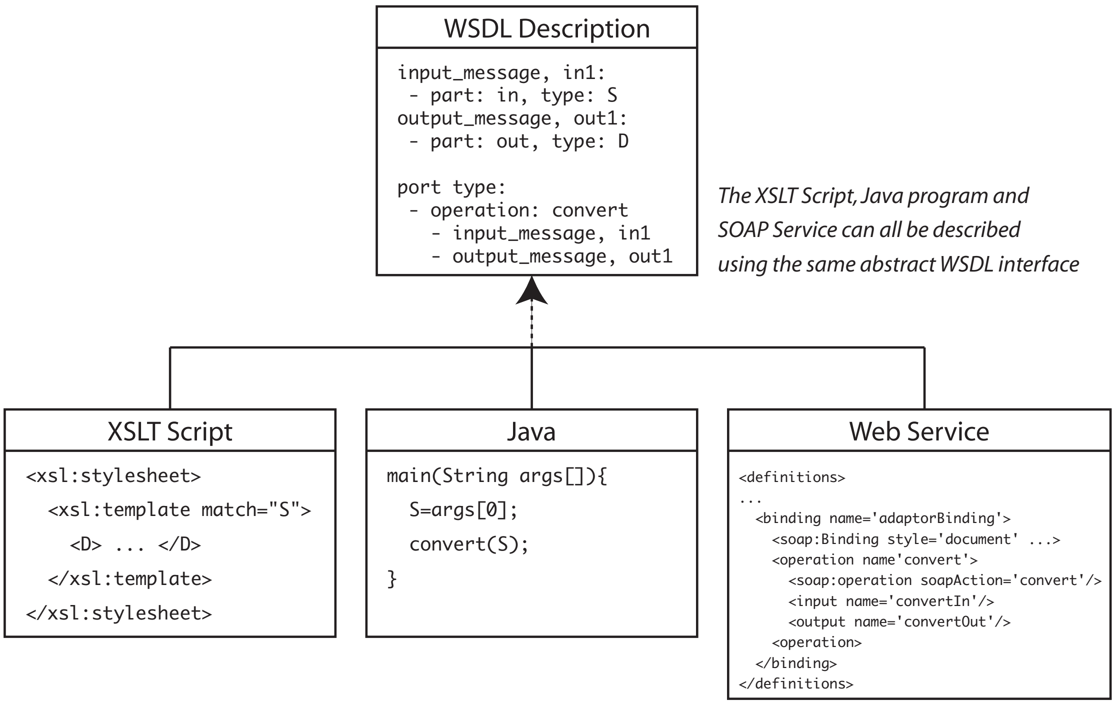
   With standardised definitions in place, Type Adaptor descriptions can be uploaded to a registry and shared with others. Such a registry must provide a suitable query interface that supports the retrieval of adaptor descriptions based on input and output types. This way, appropriate software can identify when a syntactic mismatch occurs within a workflow and find the relevant Type Adaptor autonomously by querying the registry.

In Section 7.3.2, we present our method for describing Type Adaptor capabilities before showing an implementation to generate descriptions automatically.

7.3.2 Generic Type Adaptor Description Approach

To describe the capabilities of all Type Adaptors, irrespective of implementation, we separate concrete implementation details from the abstract definition. Under this assumption, all Type Adaptors can be described using WSDL [33].

WSDL is a declarative language used to specify service capabilities and how to access them through the definition of service end-points. The operations implemented by the service are defined in terms of the messages consumed and produced, the structure of which is specified by XML Schema. The service, operations and messages are described at an abstract level and bound to a concrete execution model via the service binding. The service binding describes the type of protocol used to invoke the service and the requested datatype encoding. Because of this two-tier model, many different Web Service implementations may be viewed through a common interface. By applying the same principle to data harmonisation components, WSDL can be used to describe the capabilities of any Type Adaptor. Using this approach allows different implementations of the same Type Adaptor to be described with the same abstract definition (i.e. in terms of the input and output XML schema types) and different bindings. This is illustrated in Figure 7.3, where three Type Adaptors are shown: an XSLT script, a JAVA program and a SOAP Web Service, all providing the same functionality - to convert data of type S to D. Although other Web technologies, such as RDF [66], would be adequate for describing Type Adaptor behaviour in this way, WSDL is standardised and widely used for other Web Service technologies (e.g. the workflow languages WSFL [68] and BPEL4WS [90], and the choreography language WS-CDL [62]), and therefore facilitates technology reuse in future work.


WSDL Description
The XSLT Script, Java program and SOAP Service can all be described using the same abstract WSDL interface
XSLT Script
Java
Web Service

```
input_message, in1:
 - part: in, type: S
output_message, out1:
 - part: out, type: D

port type:
 - operation: convert
   - input_message, in1
   - output_message, out1
```

```
<xsl:stylesheet>
  <xsl:template match="S">
    <D> ... </D>
  </xsl:template>
</xsl:stylesheet>
```

```
main(String args[]){
  S=args[0];
  convert(S);
}
```

```
<definitions>
...
  <binding name='adaptorBinding'>
    <soap:Binding style='document' ...>
    <operation name'convert'>
      <soap:operation soapAction='convert'/>
      <input name='convertIn'/>
      <output name='convertOut'/>
    <operation>
  </binding>
</definitions>
```

FIGURE 7.3: Using WSDL to describe different Type Adaptors

With a uniform method for the description of Type Adaptors in the form of WSDL, we can utilise existing registry technologies to support sharing and discovery - this feature is described in more detail in Section 7.3.4. Figure 7.4 shows a high level view of how a registry containing WSDL definitions of Type Adaptors can be used in our use case workflow to perform syntactic mediation. The output from the DDBJ Service, of XML type DDBJ, is used as input to the NCBI-Blast Service, which consumes type FASTA. The binding section of the WSDL definition describes how to execute the translator, for example, by providing the location of an XSLT script or the JAVA method details.

7.3.3 WSDL Generation for M-Bindings

Within the WS-HARMONY architecture, translation may be performed using an intermediary based adaptor which converts data from a source type to a destination type via an intermediate OWL representation. Using the mapping language FXML-M, presented in Chapter 5, and the Configurable Mediator, shown in Chapter 6, conversion between semantically equivalent data representations can be achieved using a realisation M-Binding and a serialisation M-Binding. Since we assume a

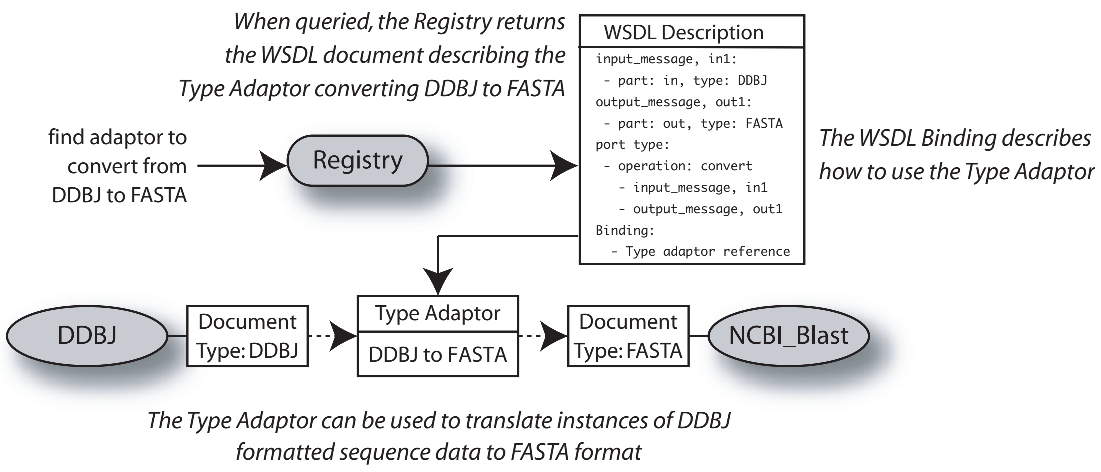
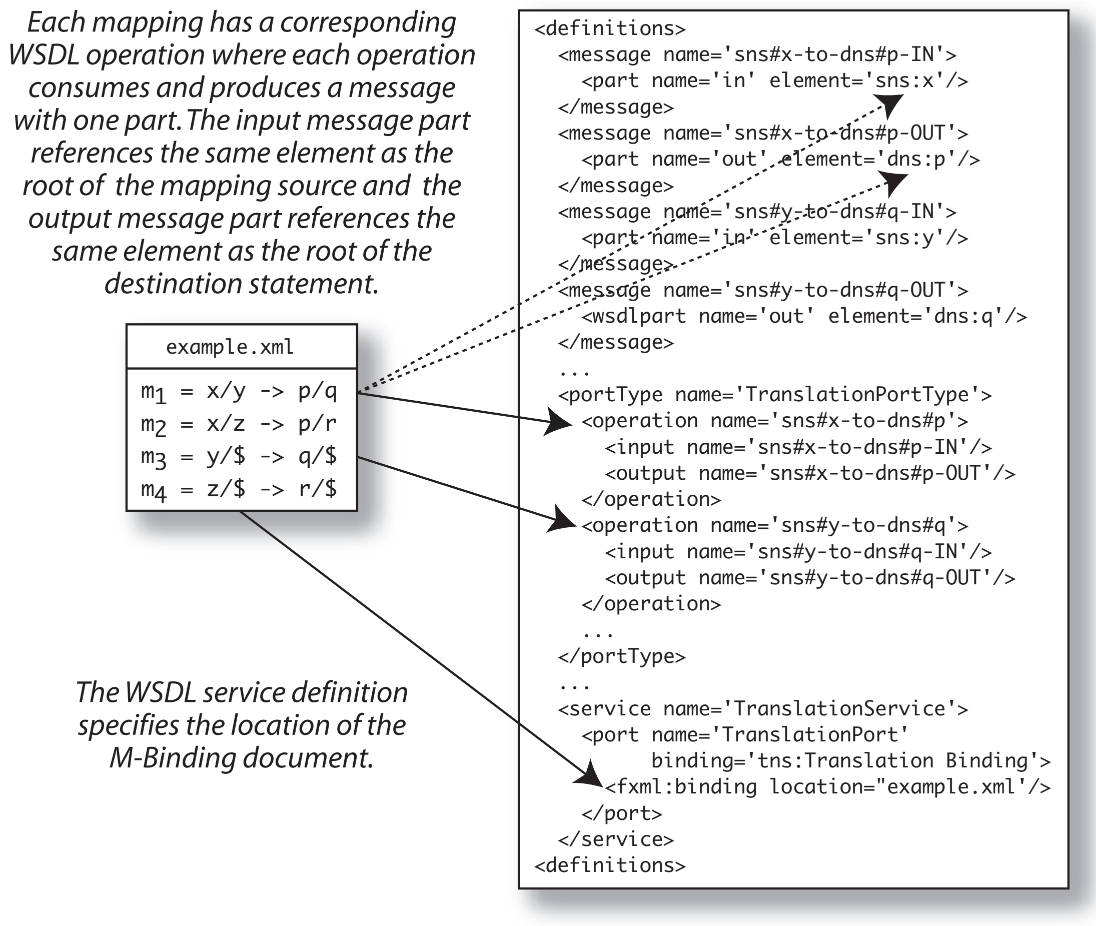

When queried, the Registry returns the WSDL document describing the Type Adaptor converting DDBJ to FASTA
find adaptor to convert from DDBJ to FASTA
Registry
WSDL Description
input_message, in1:
- part: in, type: DDBJ
output_message, out1:
- part: out, type: FASTA
port type:
- operation: convert
- input_message, in1
- output_message, out1
Binding:
- Type adaptor reference
The WSDL Binding describes how to use the Type Adaptor
DDBJ
Document Type: DDBJ
Type Adaptor
DDBJ to FASTA
Document Type: FASTA
NCBI_Blast
The Type Adaptor can be used to translate instances of DDBJ formatted sequence data to FASTA format

FIGURE 7.4: The use of a registry to discover Type Adaptors

canonical XML representation for OWL concept instances, which can be validated using automatically generated OWL-$\mathcal{XIS}$ (OWL instance schema), M-Bindings converting XML to OWL and vice-versa can be described as an adaptor converting from a source XML type to a destination XML type - i.e. the same as a direct mediation component.

For the sake of automation, we provide a system to generate WSDL definitions for M-Bindings so their descriptions can be added to a registry automatically. Since an M-Binding is a sequence of mappings, $B = \{m_1, m_2, \ldots, m_n\}$, with each mapping m_i denoting a transformation rule, a WSDL definition must capture all possible transformations catered for by B - namely an operation for each mapping m_i. When generating a WSDL definition, each mapping is given a corresponding WSDL operation that consumes an input message and produces an output message, each with one message part. The input message part references the same element as the root of the mapping source statement and the output message part references the same elements as the root of the mapping destination statement, as we show in Figure 7.5. The WSDL service definition specifies the location of the M-Binding document using the extensibility point and the `<fxml:binding location='...'>` element. This allows a user or software component to retrieve the M-Binding document when given the WSDL definition. Pseudocode for the generation of WSDL documents that describe M-Binding capability is given in Listing 7.10.



Each mapping has a corresponding WSDL operation where each operation consumes and produces a message with one part. The input message part references the same element as the root of the mapping source and the output message part references the same element as the root of the destination statement.

```
example.xml
m1 = x/y -> p/q
m2 = x/z -> p/r
m3 = y/$ -> q/$
m4 = z/$ -> r/$
```

The WSDL service definition specifies the location of the M-Binding document.

```
<definitions>
  <message name='sns#x-to-dns#p-IN'>
    <part name='in' element='sns:x'/>
  </message>
  <message name='sns#x-to-dns#p-OUT'>
    <part name='out' element='dns:p'/>
  </message>
  <message name='sns#y-to-dns#q-IN'>
    <part name='in' element='sns:y'/>
  </message>
  <message name='sns#y-to-dns#q-OUT'>
    <wsdlpart name='out' element='dns:q'/>
  </message>
  ...
  <portType name='TranslationPortType'>
    <operation name='sns#x-to-dns#p'>
      <input name='sns#x-to-dns#p-IN'/>
      <output name='sns#x-to-dns#p-OUT'/>
    </operation>
    <operation name='sns#y-to-dns#q'>
      <input name='sns#y-to-dns#q-IN'/>
      <output name='sns#y-to-dns#q-OUT'/>
    </operation>
    ...
  </portType>
  ...
  <service name='TranslationService'>
    <port name='TranslationPort'
          binding='tns:Translation Binding'>
      <fxml:binding location="example.xml'/>
    </port>
  </service>
<definitions>
```

FIGURE 7.5: The relationship between and M-Binding and its WSDL definition

After setting the target namespace of the WSDL to the same as the M-Binding (line 2), a new service element is created (line 3) using the location of the M-Binding, a portType is added (line 4), and the source and destination schemas are imported (lines 5 and 6). For the generation, an iteration is made through all mappings in the global scope, adding an input message and an output message for each. The input message type (with the part name "IN") is the same as the first component referenced in the source mapping path (line 14). The output message type (with the part name "OUT") is the same as the first component referenced in the destination mapping path (line 15). Once the message have been created, an operation can be added to the portType (line 17).

Figure 7.6 illustrates our *Binding Publisher Service* which can be used to automatically generate WSDL definitions of M-Bindings and publish them with the

```
create-wsdl(mbinding, wsdl){
  wsdl.setTargetNS(mbinding.targetNS)
  wsdl.addservice(mbinding.location)
  portType <- wsdl.addPortType("TranslationPortType")
  wsdl.importType(mbinding.sourceNS)
  wsdl.importType(mbinding.destinationNS)

  foreach mapping in mbinding
    inmessage <- wsdl.addMessage("sns#" + mapping.sourceroot.localname + "-to-dns# +
                               mapping.destinationRoot.localname + "-IN")

    outmessage <- wsdl.addMessage("sns#" + mapping.sourceroot.localname + "-to-dns# +
                                mapping.destinationRoot.localname + "-OUT")
    inmessage.addpart("in", mapping.sourceRoot)
    outmessage.addpart("out", mapping.destinationRoot)

    portType.addOperation("sns# + mapping.sourceroot.localname + "-to-dns#" +
                         mapping.destinationRoot.localname, inmessage, outmessage)
  RETURN wsdl
}
```

LISTING 7.10: Pseudocode for the generation of WSDL definitions that capture *M*-Binding capability.

GRIMOIRES registry. The Binding Publisher Service takes three inputs: an *M*-Binding, a source XML schema and a destination XML schema (1). After generating the WSDL description (2), it is published in the GRIMOIRES repository (3) so it can be found at later time using the standard GRIMOIRES API call `findinterface` (4). The WSDL document returned contains the location of the *M*-Binding document

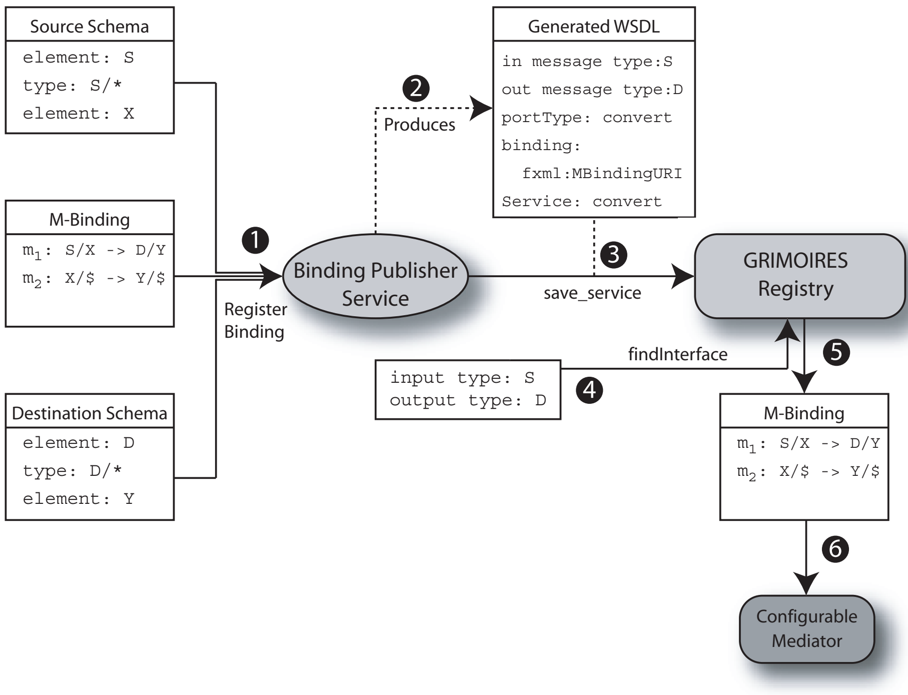


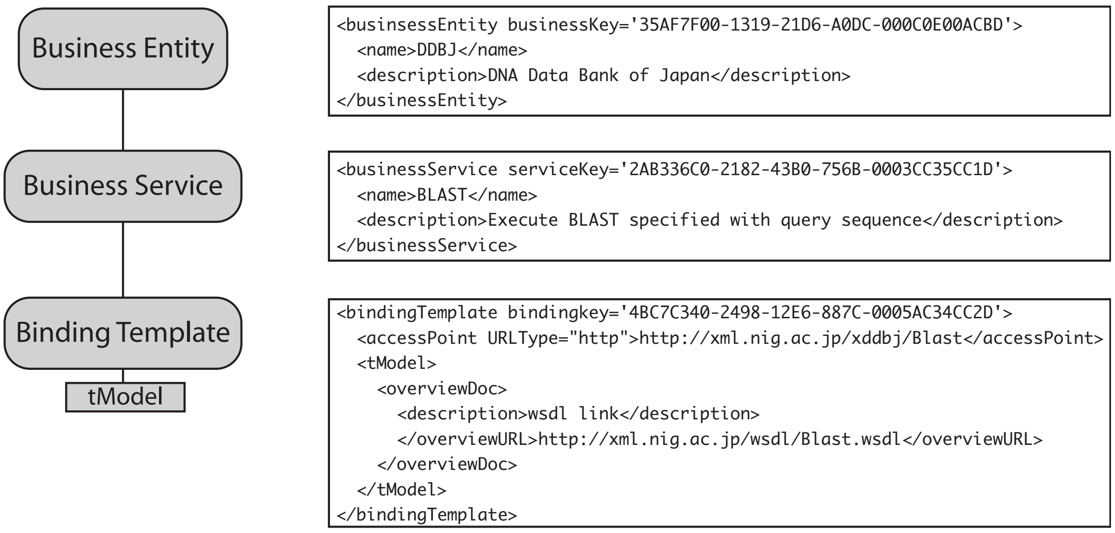
FIGURE 7.6: The Binding Publisher Service can be used to automatically generate WSDL definitions of *M*-Bindings and register them with GRIMOIRES

(5) which can then be consumed by the C-Mediator (6) to drive translation.

7.3.4 Grimoires Repository

To support the sharing and discovery of Type Adaptor descriptions, we utilise the Grimoires (www.grimoires.org) registry. Grimoires is an extended uddi [1] registry that supports publishing, annotation and discovery of service interfaces. uddi, the Web Services standard for interface publication, enables service providers to advertise service descriptions through the use of a standardised model. This model is broadly broken into three tiers, illustrated in Figure 7.7:

1. **Business Entity:** The top level container that holds description information about a business or entity. Each service provider is allocated a unique business entity ID to which they can add business services.
2. **Business Service:** Each service offered by a business entity is allocated a unique business service ID. A Business entity can provide multiple services.
3. **Binding Template:** For each business service, a binding template is created to specify the actual end point of their service, for example, the wsdl document location. This information is encapsulated with a tModel data structure.



```
<businsessEntity businessKey='35AF7F00-1319-21D6-A0DC-000C0E00ACBD'>
  <name>DDBJ</name>
  <description>DNA Data Bank of Japan</description>
</businessEntity>
```

```
<businessService serviceKey='2AB336C0-2182-43B0-756B-0003CC35CC1D'>
  <name>BLAST</name>
  <description>Execute BLAST specified with query sequence</description>
</businessService>
```

```
<bindingTemplate bindingkey='4BC7C340-2498-12E6-887C-0005AC34CC2D'>
  <accessPoint URLType="http">http://xml.nig.ac.jp/xddbj/Blast</accessPoint>
  <tModel>
    <overviewDoc>
      <description>wsdl link</description>
      </overviewURL>http://xml.nig.ac.jp/wsdl/Blast.wsdl</overviewURL>
    </overviewDoc>
  </tModel>
</bindingTemplate>
```

Figure 7.7: An overview of the uddi data model with examples in xml

Because UDDI only provides a contact point for service descriptions, it lacks the ability to support the discovery of services according to interface properties such as the input or output message parts. GRIMOIRES has been developed to solve this problem, providing an extended UDDI registry offering two notable features:

- **Meta-data annotation**
  By storing all UDDI models and WSDL definitions in RDF, GRIMOIRES supports arbitrary annotation of interface definitions. For example, any part of the WSDL definition can be linked to a concept in an ontology to give WSDL message parts a semantic type or classify a WSDL operation. GRIMOIRES provides a meta-data query interface so services can be discovered according to their meta-data attachments.

- **WSDL query interface**
  Given the wide use of WSDL, and the inability of conventional UDDI registries to support the retrieval of services according to WSDL features, GRIMOIRES offers a WSDL query interface that enables searching over WSDL features.

To use the GRIMOIRES registry for the advertising and discovery of Type Adaptors, we create three business entities: one to hold direct mediation definitions, and another two to hold intermediary based mediation definitions (one for conceptual serialisation and one for conceptual realisation). This separation is used so users or software components can query for specific types of mediators, ensuring that other services are excluded from the search. In Figure 7.8, we show how GRIMOIRES can be used in our use case scenario to find the necessary M-Bindings to perform syntactic mediation via an intermediate OWL representation. The output of the DDBJ-XML service, of type DDBJ, is not suitable for input to the NCBI-Blast service because it consumes FASTA format. Since both data types have been assigned the same semantic type (the `Sequence_Data` concept), an OWL concept instance can be used as the intermediate representation. Therefore, two queries are sent to the GRIMOIRES repository: one for a Type Adaptor that converts from DDBJ to `Sequence_Data` , and another that converts from `Sequence_Data` to FASTA format. The WSDL documents returned from this query point to the relevant *M*-Bindings so they can be consumed by the C-MEDIATOR to drive translation.

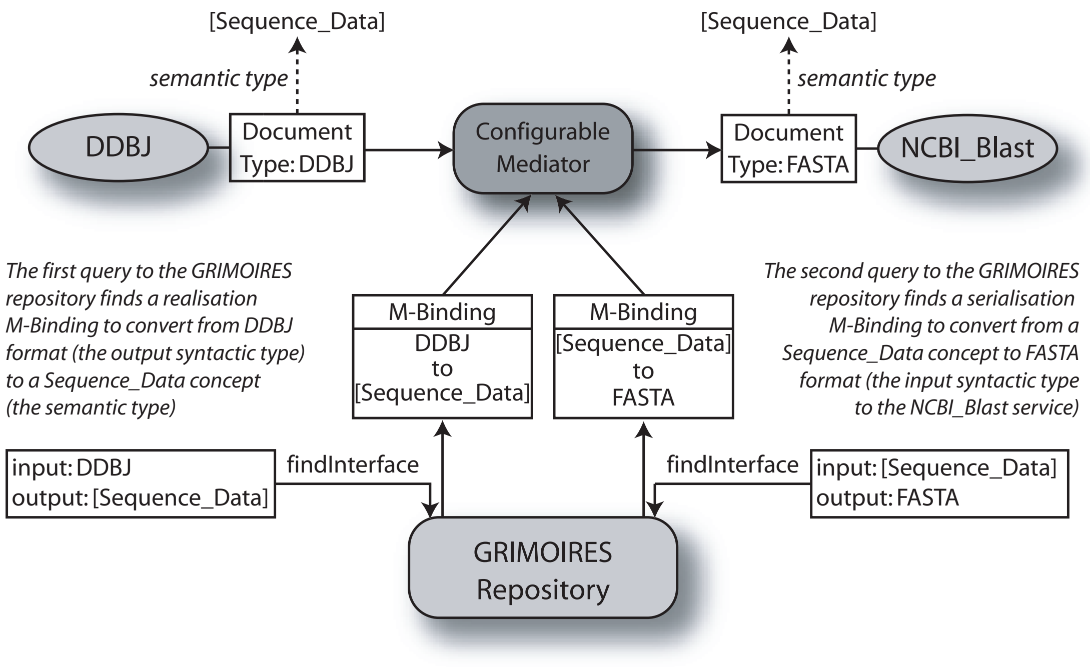

[Sequence_Data]
semantic type
DDBJ
Document Type: DDBJ
Configurable Mediator
[Sequence_Data]
semantic type
Document Type: FASTA
NCBI_Blast
The first query to the GRIMOIRES repository finds a realisation M-Binding to convert from DDBJ format (the output syntactic type) to a Sequence_Data concept (the semantic type)
M-Binding DDBJ to [Sequence_Data]
M-Binding [Sequence_Data] to FASTA
The second query to the GRIMOIRES repository finds a serialisation M-Binding to convert from a Sequence_Data concept to FASTA format (the input syntactic type to the NCBI_Blast service)
input: DDBJ output: [Sequence_Data]
findInterface
findInterface
input: [Sequence_Data] output: FASTA
GRIMOIRES Repository

FIGURE 7.8: How the GRIMOIRES repository can be used to discover M-Bindings at run time

7.4 Evaluation

To evaluate the middleware components we have presented in this Chapter, we perform two tests: (i) to check the performance of the DWSI; and (ii) to ensure that the discovery of M-Binding documents is not significant compared to the cost of invoking the target services. The test setup is the same as was specified earlier in Chapter 6, Section 6.5.

7.4.1 Dynamic WSDL Invoker

We test the performance of the DWSI by invoking the DDBJ-XML web service multiple times to retrieve random Sequence Data Records with a range of sizes from 2KBytes to 140KBytes. For comparison, we test the DWSI against the JAVA based Apache Axis toolkit. The test hypothesis follows:

H4. The Dynamic Web Service Invoker performs well in comparison to other invocation frameworks and scales linearly as input or output document size is increased.

Figure 7.9 is a graph that shows invocation time (in milliseconds) against the size of Sequence Data Record retrieved. For relatively small output documents, around 20KBytes, the DWSI and Apache Axis implementations are roughly the same. However, as the document size increases, the DWSI is able to retrieve the document between 30% and 50% quicker than Apache Axis. During the invocation of a Web Service, a significant amount of time is spent sending the SOAP envelope over the network, waiting for the service to respond, and receiving the response envelope. When the message size is fairly small, the time taken by each implementation to create the envelope, either by parsing the XML document in the case of the DWSI or serialising the JAVA objects for Apache Axis, is relatively small in comparison. However, as the output document size increases, the SOAP envelope creation time is more significant. The times recorded in this test indicate the point where either the XML output document is created (for the DWSI), or the JAVA objects are instantiated in memory (for Apache Axis). In the WS-HARMONY architecture, the output of the service may be passed to a C-MEDIATOR for translation. When this occurs, the C-MEDIATOR can directly consume the output XML document. If Apache Axis was used, a further processing step would be required to convert the JAVA objects to an XML representation.

7.4.2 Discovery Cost

To evaluate our discovery implementation, we consider the relative cost of using GRIMOIRES to discover M-Bindings in the context of workflow execution. The hypothesis is as follows:

H5. The cost of M-Binding discovery using GRIMOIRES *is not significant when compared to the cost of executing the target services.*

We test our hypothesis against our use case workflow using the DDBJ-XML and NCBI-Blast services. The Table below shows the average time taken (from 10 runs) for each step of the mediation process using OWL as an intermediary representation. The translation process is broken into 5 steps:

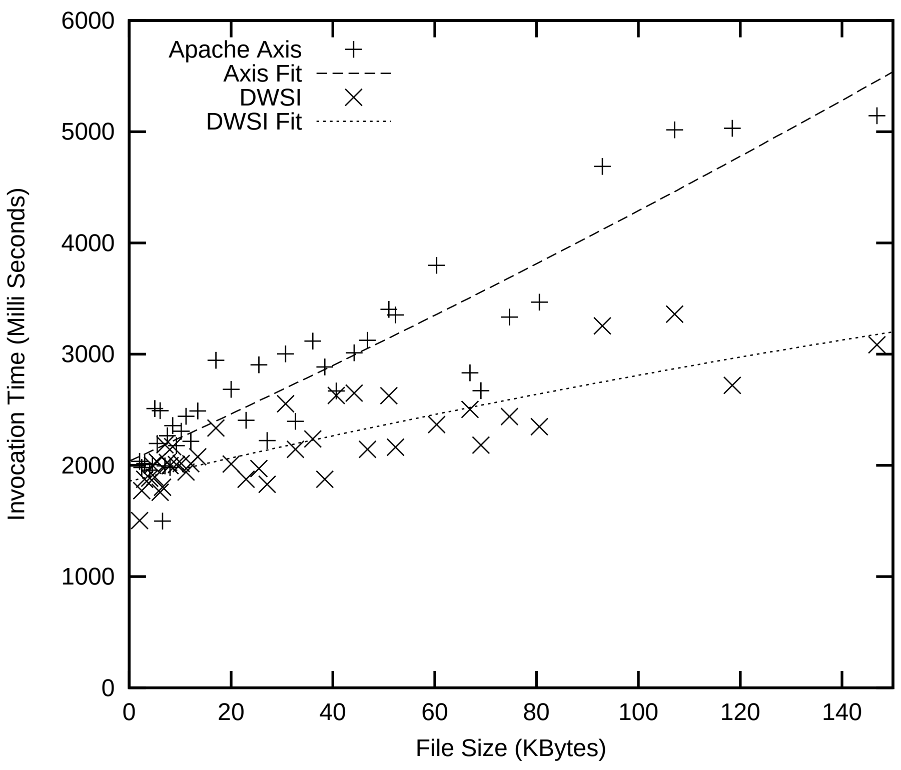


FIGURE 7.9: DWSI and Apache Axis performance invoking the DDBJ-XML Web Service

1. **Discover realisation M-Binding**

   The DWSI is used to query the GRIMOIRES repository for a Type Adaptor that converts from DDBJXML to `Sequence_Data_Record`.

2. **Conceptual Realisation**

   The DDBJXML record is transformed to an instance of the `Sequence_Data` concept.

3. **Modelling of OWL concept instance**

   The `Sequence_Data_Record` concept instance is imported into JENA.

4. **Discover serialisation M-Binding**

   The DWSI is used to query the GRIMOIRES repository for a Type Adaptor that converts from `Sequence_Data_Record` to FASTA.

5. **Conceptual Serialisation**

   The `Sequence_Data_Record` concept instance is transformed to FASTA format by the Translation Engine.

Activity	*Average*
DDBJ Execution	2.50
Realisation Discovery 1.	0.22
Realisation Transformation 2.	0.47
Jena Mediation 3.	0.62
Serialisation Discovery 4.	0.23
Serialisation Translation 5.	0.27
Total Mediation	1.81

Results show that the total mediation time is just under 2 seconds, with the largest portion of the time taken importing the OWL instance into JENA. The discovery overhead (finding realisation and serialisation M-Bindings) is small in comparison, 0.22 seconds and 0.23 seconds respectively, which totals 20% of the time taken to execute the DDBJ-XML service. Other services, such as the NCBI-Blast service, can take much longer to execute — times in excess of 1 minute are not uncommon — so discovery time within this context is low. Although GRIMOIRES implements UDDI, our discovery mechanism requires the use of additional GRIMOIRES functionality, namely, the retrieval of service based on their input and output types. This is not implemented in UDDI but can be achieved with GRIMOIRES using meta data attachment. Fang *et al* [42] show that GRIMOIRES discovery time scales well as more descriptions are added, so we infer that our discovery process comes with an acceptable performance cost.

7.5 Conclusions

In this Chapter, we have presented the middleware components of the WS-HARMONY architecture that enable the invocation of WSDL services, generation of OWL-$\mathcal{XIS}$, and the discovery of Type Adaptor specifications. Our Dynamic Web Service Invoker provides an effective way to invoke previously unseen WSDL services that would otherwise be problematic using existing Web Service invocation APIs. The OWL-$\mathcal{XIS}$ provides the bridge between OWL ontologies and their corresponding serialisations in XML, supporting the specification of mappings between XML schemas and OWL ontologies. By using WSDL to describe adaptor capabilities, both direct and intermediary based mediators can be shared among users, reducing effort in the development of adaptor components and facilitating the autonomous discovery of harmonisation components. By automatically generating WSDL descriptions of M-Binding capabilities and registering them with the GRIMOIRES repository, the C-MEDIATOR can find serialisation and realisation M-Bindings at run time, providing an automatic harmonisation infrastructure that we demonstrate against a bioinformatics use case. Empirical testing shows that the discovery process comes with a relatively low cost in comparison to the execution of target services, and would scale well as more descriptions are added providing an efficient registry implementation, such as GRIMOIRES, is used. Caching mechanisms that track the discovery of Type Adaptors could be implemented to improve discovery performance and would be useful if particular adaptors are searched for more than others. Using a logical separation between Type Adaptors and normal services within the repository, through the use of UDDI business entities, means queries for adaptor components will not return other sorts of service that could effect the meaning of the workflow.

Chapter 8

Conclusions and Future Work

In scientific, service oriented environments, where access to a variety of data repositories and computational analysis tools is exposed via Web Services, scientists rely on the similarity between workflow design and experiment design to perform *in silico* science. Users decompose their experimental processes into a set of tasks, then discover service instances to realise them, mapping the process control onto a workflow over these service instances. With the recent inclusion of semantic service annotations, the service discovery process has evolved: instead of searching over interface definitions alone (which are often terse and undocumented), users can find the services they need by specifying the functional requirements of a service using terminology from a domain ontology. After finding service instances to fulfil the tasks within their experimentation process, the user creates a workflow to control the order of execution and the flow of data between services. However, Chapter 2 showed that workflow design is often complicated because service providers can assume different representations for conceptually equivalent data. This confuses users because semantically interoperable service interfaces, i.e. those which produce and consume information that is assigned the same concept from an ontology, may be syntactically incompatible. The current solutions to this problem require the manual insertion of Type Adaptor components to perform the necessary syntactic mediation, effectively enforcing workflow harmonisation on the workflow designer. The result is a convoluted workflow design pattern in which users have to consider not only the scientific aims of their design, but also the low-level interoperability issues between services. Consequently, this distracts users from the real scientific problem they want to address and reduces accessibility to non-technical users.

Through an investigation of related work in Chapter 3, we discovered that a combination of Semantic Web Service technology with existing data integration techniques can yield solutions that support users in the creation of meaningful workflows without concern for the interoperability issues that arise from heterogeneous data representations. Such a solution is presented in Chapter 4 in the form of our Web Service mediation framework, WS-HARMONY. From a global perspective, we separate the mediation of data into two categories: *direct*, where transformation is performed straight from one format to another; and *intermediary-based*, where a common data model is used to mediate between the two formats. With a direct approach, scalability is poor; as the number of compatible data formats increases, the number of translation components required is $O(n^2)$. When introducing a new data format for which there already exists conceptually equivalent formats, translation components must be written from the new format to all existing formats to achieve maximum interoperability. With an intermediary-based approach scalability is much better; a constant increase in the number of translation components will occur as the number of compatible data formats is increased - $O(n)$. In addition, the introduction of new data formats is made easier because only a translation to and from the intermediary format is required. While we focus our efforts on an intermediary-based approach, discovery of direct Type Adaptors is supported to cater for the conversion components that already exist within MYGRID.

Intermediary-based mediation within the WS-HARMONY architecture is supported using OWL ontologies that capture the structure and semantics of data formats, with mappings that specify how data instances are transformed to and from a conceptual representation. WS-HARMONY uses an OWL concept instance as an intermediate representation to translate conceptually equivalent data between different syntactic formats. The transformation of data is handled by the Configurable Mediator (C-MEDIATOR) - a software component that consumes a mapping, a source data instances, schemas for the source and destination data format, and an ontology definition in OWL, and produces a data instance in the destination format. Because service providers often expose many operations that consume and produce information over the same, or subsets of the same, data format, we champion a mapping approach that is both modular and composable to facilitate the reuse of mapping definitions. In terms of the mapping specification, it is beneficial to de-couple it from the service interface definition, so that service providers can continue to expose access to their resources in the conventional manner without having to add mapping definitions. Also, when multiple operations are exposed over the same, or subsets of the same data type, only a single mapping definition for that type is needed, rather than one for each operation.

To express the relation between XML schema components and OWL concepts, we define the mapping language FXML, presented in Chapter 5. Examination of the data formats from our use case shows that the translation between XML data sources and their corresponding conceptual models in OWL is often complex when considered from a modular perspective. Therefore, we developed a formalisation to express the mapping of schema components and the translation process between data formats. This low-level approach has allowed us to understand and capture the complex translation requirements, notably document paths, predicate-based evaluation, local scoping and string manipulation, as well as providing a solid foundation on which we built our transformation engine FXML-T. Chapter 6, presents the implementation of the C-MEDIATOR, with particular emphasis on the translation engine FXML-T. Through empirical testing, we show that the implementation is scalable with respect to increasing document sizes and increasing schema size, as well demonstrating that binding composition comes with virtually zero cost.

Automated workflow harmonisation: the discovery of appropriate mappings on behalf of the user at runtime, can be achieved using a registry that supports the advertising and discovery Type Adaptors based on conversion capabilities. Chapter 7 presented a method to describe the capabilities of Type Adaptors in such a way that they may be discovered according to their functionality by using the Web Services Description language WSDL. Because WSDL separates the abstract functionality of a software component from the implementation specifics, Type Adaptors can be described and discovered in terms of their conversion capabilities without consideration for implementation. Using WSDL means translation specifications, such as XSLT scripts and M-Bindings, as well as applications, such as JAVA programs and web services, can all be specified using WSDL with the binding portion of a WSDL document giving the appropriate instructions on how to invoke the Type Adaptor. The WS-HARMONY architecture uses the GRIMOIRES grid registry for the advertising and discovery of Type Adaptors and provides a registration service that automatically generates WSDL descriptions for M-Binding documents. Existing GRIMOIRES API calls are used to support the discovery of Type Adaptors by the input type and desired output type. This approach works for both direct and intermediary-based adaptors so existing conversion components can be shared easily amongst users.

In general, the contributions of this dissertation can be considered a fundamental step towards the realisation of a Semantic Web Services vision. While a significant portion of research in this area has focused on the methods for capturing the meaning of service interfaces and how to orchestrate their coordination, the relationship between high-level descriptions and low-level interface definitions has been largely overlooked, a problem exemplified in this dissertation.

8.1 Future Work

The contributions of this dissertation can be used to further the state of the art in the following ways:

8.1.1 Semantic Workflow

Much effort has been placed into the research and development of workflow languages that enable the specification of complex tasks over multiple providers at a high level of abstraction [73, 38, 31, 30]. While current workflow languages support the amalgamation of computing assets to meet intricate user requirements, a considerable amount of technical knowledge is still required to create stable and functioning workflows. Enabling scientists to express the requirements of their experimentation process at a high level of abstraction using intuitive process control requires even more complex middleware. For example, when dataflow

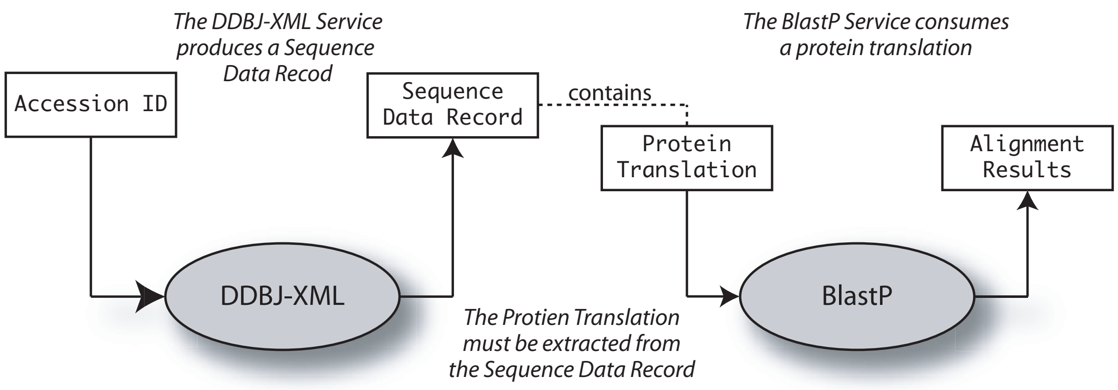

The DDBJ-XML Service produces a Sequence Data Recod
The BlastP Service consumes a protein translation
Accession ID
Sequence Data Record
contains
Protein Translation
Alignment Results
DDBJ-XML
BlastP
The Protien Translation must be extracted from the Sequence Data Record

FIGURE 8.1: An example showing non-trivial data flow between semantically annotated Web Services.

between two services is used, it is often the case that only a subset of the information from the source service is required for input to the destination service. This can be illustrated in terms of our use case easily because many services operate over subsets of a sequence data record. Sequence data records that describe proteins contain a translation of the DNA sequence to a protein sequence (e.g. `atgagtgatggagcagttcaaccagacggtggtcaacctg` is translated to `MSDGAVQPDG`). This protein sequence itself can then be passed to a computational analysis tool such as BlastP, illustrated in Figure 8.1. To hide the fact that a part of the sequence data record must be extracted (and possibly transformed to another representation), existing middleware must be augmented. In this example, our mapping technology could be reused easily to support the extraction of data. In other cases, where large sets or lists of records are produced by services (e.g. Blast results), feeding the output to another service which consumes only single records requires more data manipulation.

8.1.2 Formal Mapping Analysis

Our XML mapping and translation formalism, FXML-M, has been used to specify how mappings between XML schema components can be used to drive the translation of XML documents. The FXML-M formalism could be extended in two ways to provide some notion of binding validity:

1. **Binding Completeness**

   When mapping XML components from a particular XML schema, it would be valuable to know that every possible combination of document that validates against that schema would be successfully transformed by a binding and that all elements would be mapped to the destination document. In cases where not all components are mapped, it would be useful to know which components would be omitted.

2. **Binding Validation**
   The current FXML-M specification makes no checks that bindings produce valid documents. While it is possible to use an XML validator to check a transformed document against its schema, it could be more cost effective to check that the binding produces a valid document before attempting to use it in a translation. This is pertinent in a scientific environment where data instances can be very large and translation would be an expensive process.

Since FXML-M covers a large number of constructs from XPATH, FXML-M could be used as a basis to formalise XSLT and XQUERY.

8.1.3 Automatic Mapping Generation

Our workflow harmonisation solution relies on mappings that convert data to and from a shared conceptual model. The binding creation process is time consuming, error prone and requires a good understanding of both XML and OWL. The ability to automatically generate these bindings would be of great value, but it is not a trivial task. Other research [7, 8, 39] has investigated this idea in the context of traditional data integration, using a combination of linguistic analysis, structural comparison and loosely defined documentation to generate mappings without human intervention. In some cases these approaches are still infeasible and some high level correspondence between elements can be used in combination with other techniques to generate mappings.

8.1.4 Ontology Mapping

To successfully integrate semantically equivalent but heterogeneous data formats, a single ontology definition is required to encapsulate the data contained within each format. While this approach works well for small scale and manageable applications, it does not scale well, not necessarily in a performance sense, but more from an engineering perspective: it is difficult to get large and disparate communities of people to agree on singular conceptual model. It is more realistic to assume that different conceptual models would evolve and themselves would require some integration. Our transformation technology could prove to be fruitful in this research area where differently structured models need to be converted to between different representations.

Appendix A

Sequence Data Record Ontology Definition

In this Appendix, we provide full OWL listings for the Sequence Data Ontology used in our use case.

```
<rdf:RDF
    xmlns="http://jaco.ecs.soton.ac.uk/ont/sequencedata#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xml:base="http://jaco.ecs.soton.ac.uk/ont/sequencedata">

  <owl:Class rdf:ID="Sequence_Data_Record">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="has_sequence"/>
        </owl:onProperty>
        <owl:cardinality
          rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality
          rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="accession_id"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:DatatypeProperty rdf:about="#accession_id">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Sequence_Data_Record"/>
  </owl:DatatypeProperty>
```

```
<owl:DatatypeProperty rdf:about="#description">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="has_reference">
  <rdfs:range rdf:resource="#Reference"/>
  <rdfs:domain rdf:resource="#Sequence_Data_Record"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#has_sequence">
  <rdfs:domain rdf:resource="#Sequence_Data_Record"/>
  <rdfs:range rdf:resource="#Sequence"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#has_feature">
  <rdfs:domain rdf:resource="#Sequence_Data_Record"/>
  <rdfs:range rdf:resource="#Feature"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="DDBJ_Sequence_Data_Record">
  <rdfs:subClassOf rdf:resource="#Sequence_Data_Record"/>
  <owl:disjointWith rdf:resource="#EMBL_Sequence_Data_Record"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="taxonomy"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="date_last_updated"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="molecular_form"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:about="#molecular_form">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#DDBJ_Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#taxonomy">
  <rdfs:domain rdf:resource="#DDBJ_Sequence_Data_Record"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

```
<owl:DatatypeProperty rdf:about="#date_last_updated">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#DDBJ_Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="EMBL_Sequence_Data_Record">
  <owl:disjointWith rdf:resource="#DDBJ_Sequence_Data_Record"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Sequence_Data_Record"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:about="#data_class">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#EMBL_Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#date_created">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#EMBL_Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#release_created">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#EMBL_Sequence_Data_Record"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Reference"/>

<owl:DatatypeProperty rdf:ID="author">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Reference"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="journal">
  <rdfs:domain rdf:resource="#Reference"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="title">
  <rdfs:domain rdf:resource="#Reference"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="has_reference_location">
  <rdfs:range rdf:resource="#Location"/>
  <rdfs:domain rdf:resource="#Reference"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Location"/>

<owl:DatatypeProperty rdf:ID="start">
  <rdfs:domain rdf:resource="#Location"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="end">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Location"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Feature">
  <owl:disjointWith rdf:resource="#Reference"/>
</owl:Class>
```

```
<owl:ObjectProperty rdf:ID="has_position">
  <rdfs:range rdf:resource="#Location"/>
  <rdfs:domain rdf:resource="#Feature"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Feature_Source">
    <rdfs:subClassOf rdf:resource="#Feature"/>
    <owl:disjointWith rdf:resource="#Feature_CDS"/>
</owl:Class>

<owl:DatatypeProperty rdf:ID="organism">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Feature_Source"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="isolate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Feature_Source"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="lab–host">
  <rdfs:domain rdf:resource="#Feature_Source"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Feature_CDS">
  <rdfs:subClassOf rdf:resource="#Feature"/>
  <owl:disjointWith rdf:resource="#Feature_Source"/>
</owl:Class>

<owl:DatatypeProperty rdf:ID="translation">
  <rdfs:domain rdf:resource="#Feature_CDS"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="protein–id">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Feature_CDS"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="product">
  <rdfs:domain rdf:resource="#Feature_CDS"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Sequence"/>

<owl:DatatypeProperty rdf:ID="data">
  <rdfs:domain rdf:resource="#Sequence"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="length">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Sequence"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="has_base_count">
  <rdfs:range rdf:resource="#Base_count"/>
  <rdfs:domain rdf:resource="#Sequence"/>
</owl:ObjectProperty>
```

```
  <owl:DatatypeProperty rdf:ID="type">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Sequence"/>
  </owl:DatatypeProperty>

  <owl:Class rdf:ID="Base_count"/>

  <owl:DatatypeProperty rdf:ID="A">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Base_count"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="T">
    <rdfs:domain rdf:resource="#Base_count"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="C">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Base_count"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="G">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Base_count"/>
  </owl:DatatypeProperty>

</rdf:RDF>
```

Listing A.1: OWL Definition for a Sequence Data Records Ontology

Appendix B

Example Mappings

This appendix contains example mappings to convert a DDBJ-XML sequence data record to an instance of the *Sequence_Data_Record* concept. Mathematical notation is given first, followed by the XML representation in *M*-Binding format.

$$
\begin{aligned}
m_1 &= \langle\, \langle \text{DDBJXML}, \text{ACCESSION} \rangle \ ,\ \langle [\text{Sequence\_Data\_Record} \times \textit{join}], [\text{accession\_id} \times \textit{branch}] \rangle \ , \emptyset \rangle \\
m_2 &= \langle\, \langle \text{ACCESSION}, \textit{value} \rangle \ ,\ \langle [\text{accession\_id} \times \textit{join}], \textit{value} \rangle \ , \emptyset \rangle \\
m_3 &= \langle\, \langle \text{DDBJXML}, \text{DEFINITION} \rangle \ ,\ \langle [\text{Sequence\_Data\_Record} \times \textit{join}], [\text{definition} \times \textit{branch}] \rangle \ , \emptyset \rangle \\
m_4 &= \langle\, \langle \text{DEFINITION}, \textit{value} \rangle \ ,\ \langle [\text{definition} \times \textit{join}], \textit{value} \rangle \ , \emptyset \rangle \\
m_7 &= \langle\, \langle \text{source}, \text{location} \rangle \ ,\ \langle [\text{Feature\_Source} \times \textit{join}], [\text{has\_position} \times \textit{branch}], [\text{Location} \times \textit{branch}] \rangle \ , \emptyset \rangle \\
m_9 &= \langle\, \langle \text{location}, \textit{value}\{\text{``\^{}[\^{}.]+''}\} \rangle \ ,\ \langle [\text{Location} \times \textit{join}], [\text{start} \times \textit{branch}], \textit{value} \rangle \ , \emptyset \rangle \\
m_{10} &= \langle\, \langle \text{location}, \textit{value}\{\text{``[\^{}.]+''}\} \rangle \ ,\ \langle \text{Location} \times \textit{join}], [\text{end} \times \textit{branch}], \textit{value} \rangle \ , \emptyset \rangle \\
m_{11} &= \langle\, \langle \text{DDBJXML}, \text{FEATURES}, \text{source} \rangle \ , \\
&\quad \langle [\text{Sequence\_Data\_Record} \times \textit{join}], [\text{has\_feature} \times \textit{branch}], [\text{Feature\_Source} \times \textit{branch}] \rangle \ , \emptyset \rangle \\
m_{12} &= \langle\, \langle \text{source}, [\text{qualifiers} \times \{\text{qualifiers}, \text{qualifiers/*/@name}\textit{value} = \text{``isolate''}\}] \rangle \ , \\
&\quad \langle [\text{Feature\_Source} \times \textit{join}], [\text{isolate} \times \textit{branch}] \rangle \ ,\ (m_{13}) \rangle \\
m_{13} &= \langle\, \langle \text{qualifiers}, \textit{value} \rangle \ ,\ \langle [\text{isolate} \times \textit{join}], \textit{value} \rangle \ , \emptyset \rangle \\
m_{14} &= \langle\, \langle \text{source}, [\text{qualifiers} \times \{\text{qualifiers}, \text{qualifiers/*/@name}\textit{value} = \text{``lab\_host''}\}] \rangle \ , \\
&\quad \langle [\text{Feature\_Source} \times \textit{join}], [\text{lab\_host} \times \textit{branch}\}] \rangle \ , (m_{15}) \rangle \\
m_{15} &= \langle\, \langle \text{qualifiers}, \textit{value} \rangle \ ,\ \langle [\text{lab\_host} \times \textit{join}], \textit{value} \rangle \ , \emptyset \rangle
\end{aligned}
$$

```
<?xml version="1.0"?>
<binding name="DDBJ-to-sequencedata"
         xmlns="http://jaco.ecs.soton.ac.uk/schema/binding"
         xmlns:sns="http://jaco.ecs.soton.ac.uk/schema/DDBJ"
         xmlns:dns="http://jaco.ecs.soton.ac.uk/ont/sequencedata"
         targetNamespace="http://jaco.ecs.soton.ac.uk/binding/DDBJ-to-sequencedata">
  <mapping>
    <source match="sns:DDBJXML/sns:ACCESSION"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:accession_id[branch]/"/>
  </mapping>

  <mapping>
      <source match="sns:ACCESSION/$"/>
      <destination create="dns:accession_id[join]/$"/>

  </mapping>

  <mapping>
    <source match="sns:DDBJXML/sns:DEFINITION"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:definition[branch]/"/>
  </mapping>

  <mapping>
    <source match="sns:DEFINITION/$"/>
    <destination create="dns:definition[join]/$"/>

  </mapping>

  <mapping>
    <source match="sns:DDBJXML/sns:DIVISION"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:division[branch]/"/>
  </mapping>

  <mapping>
    <source match="sns:DIVISION/$"/>
    <destination create="dns:division[join]/$"/>

  </mapping>

  <!-- Feature Location -->

  <mapping>
    <source match="sns:source/sns:location"/>
    <destination create="dns:Feature_Source[join]/dns:has_position[branch]/
                        dns:Location[branch]"/>
  </mapping>

  <mapping>

    <source match="sns:cds/sns:location"/>
    <destination create="dns:Feature_CDS[join]/dns:has_position[branch]/
                        dns:Location[branch]"/>
  </mapping>

  <mapping>
    <source match="sns:location/$^[^.]+"/>
    <destination create="dns:Location[join]/dns:start[branch]/$"/>
  </mapping>

  <mapping>
    <source match="sns:location/$[^.]+$"/>
    <destination create="dns:Location[join]/dns:end[branch]/$"/>
  </mapping>
```

```
<!-- Feature Source -->

<mapping>
  <source match="sns:DDBJXML/sns:FEATURES/sns:source"/>
  <destination create="dns:DDBJ_Sequence_Data_Record[join]/
                       dns:has_feature[branch]/dns:Feature_Source[branch]"/>
</mapping>

<mapping>
  <source match='sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "isolate"]'/>
  <destination create="dns:Feature_Source[join]/dns:isolate[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:isolate[join]/$"/>
  </mapping>
</mapping>

<mapping>
  <source match='sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "lab_host"]'/>
  <destination create="dns:Feature_Source[join]/dns:lab-host[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:lab-host[join]/$"/>
  </mapping>
</mapping>

<mapping>
  <source match='sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "mol_type"]'/>
  <destination create="dns:Feature_Source[join]/dns:molecular-type[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:molecular-type[join]/$"/>
  </mapping>
</mapping>

<mapping>
  <source match='sns:source/sns:qualifiers[sns:qualifiers/sns:name/$ = "organism"]'/>
  <destination create="dns:Feature_Source[join]/dns:organism[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:organism[join]/$"/>
  </mapping>
</mapping>

<!-- Feature CDS -->

<mapping>
  <source match="sns:DDBJXML/sns:FEATURES/sns:cds"/>
  <destination create="dns:DDBJ_Sequence_Data_Record[join]/
                       dns:has_feature[branch]/dns:Feature_CDS[branch]"/>
</mapping>

<mapping>
  <source match='sns:cds/sns:qualifiers[sns:qualifiers/sns:name/$ = "product"]'/>
  <destination create="dns:Feature_CDS[join]/dns:product[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:product[join]/$"/>
  </mapping>
</mapping>
```

```
<mapping>
  <source match='sns:cds/sns:qualifiers[sns:qualifiers/sns:name/$ = "protein_id"]'/>
  <destination create="dns:Feature_CDS[join]/dns:protein-id[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:protein-id[join]/$"/>
  </mapping>
</mapping>

<mapping>
  <source match='sns:cds/sns:qualifiers[sns:qualifiers/sns:name/$ = "translation"]'/>
  <destination create="dns:Feature_CDS[join]/dns:translation[branch]"/>
  <mapping>
    <source match="sns:qualifiers/$"/>
    <destination create="dns:translation[join]/$"/>
  </mapping>
</mapping>

<!-- Reference -->

<mapping>
  <source match="sns:DDBJXML/sns:REFERENCE"/>
  <destination create="dns:DDBJ_Sequence_Data_Record[join]/
                       dns:has_reference[branch]/dns:Reference"/>
</mapping>

<mapping>
  <source match="sns:REFERENCE/sns:authors"/>
  <destination create="dns:Reference[join]/dns:author[branch]/"/>
</mapping>

<mapping>
  <source match="sns:authors/$"/>
  <destination create="dns:author[join]/$"/>
</mapping>

<mapping>
  <source match="sns:REFERENCE/sns:title"/>
  <destination create="dns:Reference[join]/dns:title[branch]/"/>
</mapping>

<mapping>
  <source match="sns:title/$"/>
  <destination create="dns:title[join]/$"/>
</mapping>

<mapping>
  <source match="sns:REFERENCE/sns:journal"/>
  <destination create="dns:Reference[join]/dns:journal[branch]/"/>
</mapping>

<mapping>
  <source match="sns:journal/$"/>
  <destination create="dns:journal[join]/$"/>
</mapping>

<!-- Sequence Data Record Metadata -->

<mapping>
  <source match="sns:DDBJXML/sns:KEYWORDS"/>
  <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:keyword[branch]/"/>
</mapping>
```

```
  <mapping>
    <source match="sns:KEYWORDS/$"/>
    <destination create="dns:keyword[join]/$"/>
   </mapping>
  <mapping>

    <source match="sns:DDBJXML/sns:LAST_UPDATE"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:date_last_updated[branch]/"/>
  </mapping>

  <mapping>
    <source match="sns:LAST_UPDATE/$"/>
    <destination create="dns:date_last_updated[join]/$"/>
  </mapping>

  <mapping>
    <source match="sns:DDBJXML/sns:MOLECULAR_FORM"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:molecular_form[branch]/"/>
  </mapping>

  <mapping>
    <source match="sns:MOLECULAR_FORM/$"/>
    <destination create="dns:molecular_form[join]/$"/>
  </mapping>

  <mapping>
    <source match="sns:DDBJXML/sns:TAXONOMY"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:taxonomy[branch]/"/>
  </mapping>

  <mapping>
    <source match="sns:TAXONOMY/$"/>
    <destination create="dns:taxonomy[join]/$"/>
  </mapping>


  <!-- Sequence Data -->

  <mapping>
    <source match="sns:DDBJXML/sns:SEQUENCE"/>
    <destination create="dns:DDBJ_Sequence_Data_Record[join]/dns:has_sequence[branch]/
                     dns:Sequence[branch]"/>
  </mapping>

  <mapping>
    <source match="sns:SEQUENCE/$"/>
    <destination create="dns:Sequence[join]/dns:data[branch]/$"/>
  </mapping>

  <mapping>
    <source match="sns:START/$"/>
    <destination create="dns:start[join]/$"/>
  </mapping>

  <mapping>
    <source match="sns:END/$"/>
    <destination create="dns:end[join]/$"/>
  </mapping>
</binding>
```

Listing B.1: *M*-Binding document to translate DDBJ-XML documents to and OWL concept instance

Appendix C

XML Schemas

In this appendix, an XML schema is provided to validate instance from the Sequence Data Record ontology.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://jaco.ecs.soton.ac.uk/ont/sequencedata"
            targetNamespace="http://jaco.ecs.soton.ac.uk/ont/sequencedata">
    <xsd:element name="keyword" type="xsd:string"/>
    <xsd:element name="accession_id" type="xsd:string"/>
    <xsd:element name="release_created" type="xsd:string"/>
    <xsd:element name="lab-host" type="xsd:string"/>
    <xsd:element name="has_base_count" type="has_base_count-TYPE"/>
    <xsd:element name="Base_count" type="Base_count-TYPE"/>
    <xsd:element name="T" type="xsd:string"/>
    <xsd:element name="has_reference" type="has_reference-TYPE"/>
    <xsd:element name="definition" type="xsd:string"/>
    <xsd:element name="molecular_form" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="molecular-type" type="xsd:string"/>
    <xsd:element name="G" type="xsd:string"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="C" type="xsd:string"/>
    <xsd:element name="A" type="xsd:string"/>
    <xsd:element name="taxonomy" type="xsd:string"/>
    <xsd:element name="end" type="xsd:string"/>
    <xsd:element name="topology" type="xsd:string"/>
    <xsd:element name="Reference" type="Reference-TYPE"/>
    <xsd:element name="length" type="xsd:string"/>
    <xsd:element name="Sequence" type="Sequence-TYPE"/>
    <xsd:element name="INSD_Sequence_Data_Record" type="INSD_Sequence_Data_Record-TYPE"/>
    <xsd:element name="isolate" type="xsd:string"/>
    <xsd:element name="has_position" type="has_position-TYPE"/>
    <xsd:element name="has_database_reference" type="has_database_reference-TYPE"/>
    <xsd:element name="start" type="xsd:string"/>
    <xsd:element name="Feature" type="Feature-TYPE"/>
    <xsd:element name="Sequence_Data_Record" type="Sequence_Data_Record-TYPE"/>
    <xsd:element name="date_created" type="xsd:string"/>
    <xsd:element name="Feature_CDS" type="Feature_CDS-TYPE"/>
    <xsd:element name="has_feature" type="has_feature-TYPE"/>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="date_last_updated" type="xsd:string"/>
    <xsd:element name="EMBL_Sequence_Data_Record" type="EMBL_Sequence_Data_Record-TYPE"/>
    <xsd:element name="db_identifier" type="xsd:string"/>
    <xsd:element name="has_sequence" type="has_sequence-TYPE"/>
```

```
<xsd:element name="Database_Reference" type="Database_Reference-TYPE"/>
<xsd:element name="product" type="xsd:string"/>
<xsd:element name="Feature_Source" type="Feature_Source-TYPE"/>
<xsd:element name="release_last_updated" type="xsd:string"/>
<xsd:element name="has_reference_location" type="has_reference_location-TYPE"/>
<xsd:element name="journal" type="xsd:string"/>
<xsd:element name="protein-id" type="xsd:string"/>
<xsd:element name="translation" type="xsd:string"/>
<xsd:element name="Location" type="Location-TYPE"/>
<xsd:element name="DDBJ_Sequence_Data_Record" type="DDBJ_Sequence_Data_Record-TYPE"/>
<xsd:element name="db_location" type="xsd:string"/>
<xsd:element name="organism" type="xsd:string"/>
<xsd:element name="version" type="xsd:string"/>
<xsd:element name="division" type="xsd:string"/>
<xsd:element name="data" type="xsd:string"/>
<xsd:element name="type" type="xsd:string"/>

<xsd:complexType name="DDBJ_Sequence_Data_Record-TYPE">
    <xsd:complexContent>
        <xsd:extension base="Sequence_Data_Record-TYPE">
            <xsd:sequence>
                <xsd:element ref="date_last_updated"/>
                <xsd:element ref="molecular_form"/>
                <xsd:element ref="taxonomy"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="has_sequence-TYPE">
    <xsd:all>
        <xsd:element ref="Sequence"/>
    </xsd:all>
</xsd:complexType>

<xsd:complexType name="has_base_count-TYPE">
    <xsd:all>
        <xsd:element ref="Base_count"/>
    </xsd:all>
</xsd:complexType>

<xsd:complexType name="Feature_CDS-TYPE">
    <xsd:complexContent>
        <xsd:extension base="Feature-TYPE">
            <xsd:sequence>
                <xsd:element ref="product" maxOccurs="unbounded"/>
                <xsd:element ref="protein-id" maxOccurs="unbounded"/>
                <xsd:element ref="translation" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Database_Reference-TYPE">
    <xsd:sequence>
        <xsd:element ref="db_identifier" maxOccurs="unbounded"/>
        <xsd:element ref="db_location" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="EMBL_Sequence_Data_Record-TYPE">
    <xsd:complexContent>
        <xsd:extension base="Sequence_Data_Record-TYPE">
            <xsd:sequence>
                <xsd:element ref="date_created" maxOccurs="unbounded"/>
                <xsd:element ref="date_last_updated"/>
                <xsd:element ref="name" maxOccurs="unbounded"/>
                <xsd:element ref="release_created" maxOccurs="unbounded"/>
                <xsd:element ref="release_last_updated" maxOccurs="unbounded"/>
                <xsd:element ref="version" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="has_position-TYPE">
    <xsd:all>
        <xsd:element ref="Location"/>
    </xsd:all>
</xsd:complexType>
<xsd:complexType name="Sequence-TYPE">
    <xsd:sequence>
        <xsd:element ref="data" maxOccurs="unbounded"/>
        <xsd:element ref="has_base_count" maxOccurs="unbounded"/>
        <xsd:element ref="length" maxOccurs="unbounded"/>
        <xsd:element ref="type" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Base_count-TYPE">
    <xsd:sequence>
        <xsd:element ref="A" maxOccurs="unbounded"/>
        <xsd:element ref="C" maxOccurs="unbounded"/>
        <xsd:element ref="G" maxOccurs="unbounded"/>
        <xsd:element ref="T" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Reference-TYPE">
    <xsd:sequence>
        <xsd:element ref="author" maxOccurs="unbounded"/>
        <xsd:element ref="has_reference_location" maxOccurs="unbounded"/>
        <xsd:element ref="journal" maxOccurs="unbounded"/>
        <xsd:element ref="title" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Location-TYPE">
    <xsd:sequence>
        <xsd:element ref="end" maxOccurs="unbounded"/>
        <xsd:element ref="start" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Sequence_Data_Record-TYPE">
    <xsd:sequence>
        <xsd:element ref="accession_id"/>
        <xsd:element ref="definition"/>
        <xsd:element ref="division"/>
        <xsd:element ref="has_feature" maxOccurs="unbounded"/>
        <xsd:element ref="has_reference" maxOccurs="unbounded"/>
        <xsd:element ref="has_sequence"/>
        <xsd:element ref="keyword" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

```
    <xsd:complexType name="has_feature-TYPE">
        <xsd:choice>
            <xsd:element ref="Feature"/>
            <xsd:element ref="Feature_CDS"/>
            <xsd:element ref="Feature_Source"/>
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="has_reference_location-TYPE">
        <xsd:all>
            <xsd:element ref="Location"/>
        </xsd:all>
    </xsd:complexType>
    <xsd:complexType name="INSD_Sequence_Data_Record-TYPE">
        <xsd:sequence>
            <xsd:element ref="date_created" maxOccurs="unbounded"/>
            <xsd:element ref="date_last_updated"/>
            <xsd:element ref="name" maxOccurs="unbounded"/>
            <xsd:element ref="release_created" maxOccurs="unbounded"/>
            <xsd:element ref="release_last_updated" maxOccurs="unbounded"/>
            <xsd:element ref="topology" maxOccurs="unbounded"/>
            <xsd:element ref="version" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Feature-TYPE">
        <xsd:sequence>
            <xsd:element ref="has_database_reference" maxOccurs="unbounded"/>
            <xsd:element ref="has_position" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="has_reference-TYPE">
        <xsd:all>
            <xsd:element ref="Reference"/>
        </xsd:all>
    </xsd:complexType>
    <xsd:complexType name="Feature_Source-TYPE">
        <xsd:complexContent>
            <xsd:extension base="Feature-TYPE">
                <xsd:sequence>
                    <xsd:element ref="isolate" maxOccurs="unbounded"/>
                    <xsd:element ref="lab-host" maxOccurs="unbounded"/>
                    <xsd:element ref="molecular-type" maxOccurs="unbounded"/>
                    <xsd:element ref="organism" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="has_database_reference-TYPE">
        <xsd:all>
            <xsd:element ref="Database_Reference"/>
        </xsd:all>
    </xsd:complexType>
</xsd:schema>
```

Listing C.1: An XML Schema to validate instances from the Sequence Data Record ontology, created automatically by the OWL-$\mathcal{XIS}$ generator

Bibliography

[1] UDDI technical white paper, September 2000. URL: http://uddi.org/pubs/uddi-tech-wp.pdf.

[2] OWL-S: Semantic markup for web service. Technical report, The OWL Services Coalition, 2006. URL: http://www.ai.sri.com/daml/services/owl-s/1.2/overview/.

[3] activeBPEL Project Homepage. URL: http://www.activebpel.org/.

[4] Alfred V. Aho. *Algorithms for finding patterns in strings.* 1990.

[5] R. Akkiraju, J. Farrell, J.Miller, M. Nagarajan, M. Schmidt, and A. Shethand. Web service semantics - WSDL-S: W3c member submission. Technical report, 2005. URL: http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.pdf.

[6] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997. URL: citeseer.ist.psu.edu/altschul97gapped.html.

[7] Yuan An, Alexander Borgida, and John Mylopoulos. Inferring complex semantic mappings between relational tables and ontologies from simple correspondences. In *OTM Conferences (2)*, pages 1152–1169, 2005.

[8] Yuan An, John Mylopoulos, and Alexander Borgida. Building semantic mappings from databases to ontologies. In *AAAI*, 2006.

[9] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S: Web service description for the semantic web. In *Proceedings of the first International Semantic Web Conference (ISWC 02)*, 2002. URL: http://xml.coverpages.org/ISWC2002-DAMLS.pdf.

[10] Apache AXIS Project Homepage. URL: http://ws.apache.org/axis/.

[11] AstroGrid Project Homepage. URL: http://www.astrogrid.org/.

[12] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*, 2003. Cambridge University Press. ISBN 0-521-78176-0.

[13] J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the algorithmic language ALGOL 60. *Communication of the ACM*, 3(5):299–314, May 1960.

[14] Steffen Balzer and Thorsten Liebig. Bridging the Gap Between Abstract and Concrete Services – A Semantic Approach for Grounding OWL-S –. In *Proceedings of the Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, pages 16–30, Hiroshima, Japan, November 2004.

[15] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland; 2 edition, 1985.

[16] Mike Beckerle. DFDL proposal and examples. Technical report, Global Grid Forum, 2004. URL: http://forge.gridforum.org/sf/docman/do/downloadDocument/projects.dfdl-wg/docman.root.current/doc5412.

[17] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for xml, 1999. URL: citeseer.ist.psu.edu/beech99formal.html.

[18] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. e-service composition by description logics based reasoning. In *Proc. of the 2003 Description Logic Workshop (DL 2003)*, pages 75–84. CEUR Electronic Workshop Proceedings, http://ceur-ws.org/Vol-81/, 2003.

[19] Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano. Semantic integration of heterogeneous information sources. *Data Knowl. Eng.*, 36(3):215–249, 2001. ISSN 0169-023X.

[20] T. Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, pages 34 – 43, 2001.

[21] Jim Blythe, Ewa Deelman, and Yolanda Gil. Planning for workflow construction and maintenance on the grid. In *ICAPS 2003 Workshop on Planning for Web Services*, 2003.

[22] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. Xquery 1.0: An XML query langauge. Technical report, W3C, 2003. URL: http://www.w3.org/TR/xquery/.

[23] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. Technical report, W3C, 2004. URL: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[24] S. Bowers and B. Ludascher. An ontology-driven framework for data transformation in scientific workflows. In *Intl. Workshop on Data Integration in the Life Sciences (DILS'04)*, 2004. URL: http://www.sdsc.edu/ ludaesch/-Paper/dils04.pdf.

[25] Dan Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF schema. Technical report, W3C, 2004. URL: http://www.w3.org/TR/rdf-schema/.

[26] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. Msl - a model for w3c xml schema. In *WWW*, pages 191–200, 2001.

[27] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. XML schema: Formal description. Technical report, W3C, 2001. URL: http://www.w3.org/TR/2001/WD-xmlschema-formal-20010320/.

[28] P. J. Brown. The ml/i macro processor. *Commun. ACM*, 10(10):618–623, 1967. ISSN 0001-0782.

[29] Andrea Cal, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Paolo Naggar, and Fabio Vernacotola. IBIS: semantic data integration at work. *Lecture Notes in Computer Science, Advanced Information Systems Engineering*, 2681/2003:79–94, 2003.

[30] Jorge Cardoso and Amit Sheth. Semantic e-workflow composition. *J. Intell. Inf. Syst.*, 21(3):191–225, 2003. ISSN 0925-9902.

[31] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Semantic workflow interoperability. In *Extending Database Technology*, pages 443–462, 1996. URL: citeseer.ist.psu.edu/casati96semantic.html.

[32] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994. URL: citeseer.ist.psu.edu/chawathe94tsimmis.html.

[33] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1, March 2001. W3C.

[34] James Clark. XSL transformations (XSLT) version 1.0. Technical report, W3C, 1999. URL: http://www.w3.org/TR/xslt.

[35] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Technical report, W3C, 1999. URL: http://www.w3.org/TR/xpath.

[36] DDBJ Web Service. URL: http://xml.ddbj.nig.ac.jp/.

[37] Jos de Bruijin and Holger Lausen. Web service modeling language (WSML), June 2005. WSMO Working Draft.

[38] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[39] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: Discovering complex semantic matches between database schemas, 2004. URL: citeseer.ist.psu.edu/dhamankar04imap.html.

[40] Brian Eisenberg and Duane Nickull. ebXML technical architecture specification v1.0.4. Technical report, ebXML, 2001. URL: http://www.ebxml.org/specs/ebTA.pdf.

[41] David C. Fallside. XML schema part 0: Primer. Technical report, W3C, 2001. URL: http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[42] Weijian Fang, Sylvia C. Wong, Victor Tan, Simon Miles, and Luc Moreau. Performance analysis of a semantics enabled service registry. In *Proceedings of the fourth UK e-Science Programme All Hands Meeting (AHM2005)*, Nottingham, UK, 2005.

[43] Dieter Fensel, Richard Benjamins, Enrico Motta, and Bob Wielinga. UPML: A framework for knowledge system reuse. In *Proceedings of the International Joint Conference on AI (IJCAI-99), Stockholm, Sweden*, 1999. URL: ftp://ftp.aifb.uni-karlsruhe.de/pub/mike/dfe/paper/upml.ijcai.pdf.

[44] Ian Foster and Carl Kesselmann. *The Grid: Blueprint for a new computing infrastrucutre*. Morgan Kaufmann, 1999.

[45] Ian Foster, Carl Kesslemann, Jeffery M. Nick, and Steven Tuecke. The physiology of the grid, an open grid services architecture for distributed systems integration, June 2002.

[46] FreeFluo Project Homepage. URL: http://freefluo.sourceforge.net/.

[47] Gottlob Frege. *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought.* 1967.

[48] Kevin Garwood, Phillip Lord, Helen Parkinson, Norman W. Paton, and Carole Goble. Pedro ontology services: A framework for rapid ontology markup. In *Proceedings of the European Semantic Web Symposium / Conference*, Heraklion, Crete, Greece, 2005.

[49] C.A. Goble, S. Pettifer, R. Stevens, and C. Greenhalgh. Knowledge Integration: In silico Experiments in Bioinformatics. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure Second Edition*. Morgan Kaufmann, November 2003.

[50] Griphyn Project Homepage. URL: http://www.griphyn.org/.

[51] T. R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, (5):199–220, 1993.

[52] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework. Technical report, W3C, 2003. URL: http://www.w3.org/TR/soap12-part1/.

[53] Guile Project Homepage. URL: http://www.gnu.org/software/guile/guile.html.

[54] V. Haarslev and R. Moller. Racer: An owl reasoning agent for the semantic web. In *In Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems, Halifax Canada*, pages 91–95, 2003. URL: citeseer.ist.psu.edu/article/haarslev03racer.html.

[55] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. Technical report, Sun Microsystems, 2002.

[56] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab 4, R. Studer, , and E. Motta. he ontology inference layer OIL. Technical report, Vrije Universteit Amsterdam, 2000.

[57] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2nd International Semantic Web Conference (ISWC)*, 2003. URL: citeseer.ist.psu.edu/article/horrocks03reducing.html.

[58] Duncan Hull, Robert Stevens, and Phillip Lord. Describing web services for user-oriented retrieval. *W3C Workshop on Frameworks for Semantics in Web Services, Digital Enterprise Research Institute, (DERI), Innsbruck, Austria*, 2005.

[59] IBM Websphere Project Homepage. URL: http://www-306.ibm.com/software/websphere/.

[60] JENA Project Homepage. URL: http://jena.sourceforge.net/.

[61] Chris Kaler. Specification: Web services security (WS-Security). Technical report, IBM, 2002. URL: http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.

[62] Nickolas Kavantzas, David Burdet, Gregory Ritzinger, and Tony Fletcher. WS-CDL web services choreography description language version 1.0. Technical report, W3C, 2004. URL: http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[63] R. Kesley, W. Clinger, and J. Rees. Revised (5) report on the alogrithmic language scheme. *Higher-Order and Symbolic Computation*, pages 7 – 105, 1998.

[64] Hyon Hee Kim and Seung-Soo Park. Semantic integration of heterogeneous xml data sources. In *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, pages 95–107, London, UK, 2002. Springer-Verlag. ISBN 3-540-44087-9.

[65] Michel Klein, Dieter Fensel, Frank van Harmelen, and Ian Horrocks. The relation between ontologies and schema-languages: Translating oil-specifications in xml-schema. In *Proceedings of the ECAI'00 workshop on applications of ontologies and problem-solving methods*, Berlin, August 2000.

[66] Graham Klyne and Jeremy J Carroll. Resource description framework (RDF): Concepts and abstract syntax. Technical report, W3C, 2004. URL: http://www.w3.org/TR/rdf-concepts/.

[67] Paul J. Layzell. The history of macro processors in programming language extensibility. *Comput. J.*, 28(1):29–33, 1985.

[68] Frank Leymann. Web services flow language (WSFL 1.0), May 2001.

[69] Chen Li and Edward Y. Chang. Answering queries with useful bindings. *Database Systems*, 26(3):313–343, 2001. URL: citeseer.ist.psu.edu/li01answering.html.

[70] Shenping Liu, Jing Mei, Anbu Yue, and Zuoquan Lin. XSDL: Making xml semantics explicit. In *Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB2004)*, pages 64–83. Springer-Verlag, 2005.

[71] Phillip Lord, Pinar Alper, Chris Wroe, and Carole Goble. Feta: A lightweight architecture for user oriented semantic service discovery. In *The Semantic Web: Research and Applications: Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece*, pages 17 – 31, January 2005.

[72] Phillip Lord, Chris Wroe, Robert Stevens, Carole Goble, Simon Miles, Luc Moreau, Keith Decker, Terry Payne, and Juri Papay. Semantic and personalised service discovery. In W. K. Cheung and Y. Ye, editors, *Proceedings of Workshop on Knowledge Grid and Grid Intelligence (KGGI'03), in conjunction with 2003 IEEE/WIC International Conference on Web Intelligence/Intelligent Agent Technology*, pages 100–107, Halifax, Canada, 2003. Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, Canada. URL: http://www.ecs.soton.ac.uk/ lavm/papers/kggi03.pdf.

[73] B. Ludascher, I. Altintas, and A. Gupta. Compiling abstract scientific workflows into web service workflows. In *15th Intl. Conference on Scientific and Statistical Database Management*, page 251, July 2003. URL: http://kbis.sdsc.edu/SciDAC-SDM/ludaescher-compiling.pdf.

[74] Bill Meadows and Lisa Seaburg. Universal business language 1.0. Technical report, OASIS, 2004. URL: http://docs.oasis-open.org/ubl/cd-UBL-1.0/.

[75] C. N. Mooers and L. P. Deutsch. Programming languages for non-numeric processing: Trac, a text handling language. In *Proceedings of the 1965 20th national conference*, pages 229–246, New York, NY, USA, 1965. ACM Press. Chairman-R. W. Floyd.

[76] Matthew Moran. D13.5v0.1 WSMX implementation, July 2004. WSMO Working Draft.

[77] Luc Moreau, Simon Miles, Juri Papay, Keith Decker, and Terry Payne. Publishing semantic descriptions of services. Technical report, Global Grid Forum, 2003. URL: http://www.ecs.soton.ac.uk/ lavm/papers/ggf9.ps.

[78] Luc Moreau, Yong Zhao, Ian Foster, Jens Voeckler, and Michael Wilde. XDTM: the XML Dataset Typing and Mapping for Specifying Datasets. In *Proceedings of the 2005 European Grid Conference (EGC'05)*, Amsterdam, Nederlands, February 2005. URL: http://www.ecs.soton.ac.uk/ lavm/papers/egc05.pdf.

[79] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005. ISSN 1533-5399.

[80] MyGrid Project Homepage. URL: http://www.mygrid.org.uk/.

[81] NCBI Web Service. URL: http://www.ncbi.nlm.nih.gov/BLAST/.

[82] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Importing the semantic web in UDDI, 2002.

[83] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax. Technical report, W3C, 2004.

[84] Dumitru Roman, Holger Lausen, and Uwe Keller. D2v1.0. web service modeling ontology (WSMO), September 2004. WSMO Working Draft.

[85] W. Schuetzelhofer and K. Goeschka. A set theory based approach on applying domain semantics to xml structures. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 4*, page 120, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1435-9.

[86] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM Syst. J.*, 31(3):590–616, 1992. ISSN 0018-8670.

[87] Robert Stevens, Carole Goble, Norman W. Paton, Sean Bechhofer, Gary Ng, Patricia Baker, and Andy Brass. Complex Query Formulation Over Diverse Information Sources in TAMBIS. In Zoe Lacroix and Terence Critchlow, editors, *Bioinformatics: Managing Scientific Data*. Morgan Kaufmann, May 2003. ISBN 1-55860-829-X.

[88] Andrew S. Tanenbaum. A general-purpose macro processor as a poor man's compiler-compiler. *IEEE Trans. Software Eng.*, 2(2):121–125, 1976.

[89] Taverna Project Homepage. URL: http://taverna.sourceforge.net/.

[90] Satish Thatte. Business process execution language for web services version 1.1. Technical report, IBM, 2003. URL: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[91] The DARPA Agent Markup Language Homepage. URL: http://www.daml.org.

[92] H. Uitermark, P. V. Oosterom, N. Mars, and M. Molenaar. Ontology-based geographic data set integration. In *Proceedings of Workshop on Spatio-Temporal Database Management*, pages 60–79, Edinburgh, Scotland, 1999.

[93] S. C. Wong, V. Tan, W. Fang, S. Miles, and L Moreau. Grimoires: Grid registry with metadata oriented interface: Robustness, efficiency, security — work-in-progress. In *In Proceedings of Work in Progress Session in Cluster Computing and Grid (CCGrid), Cardiff, UK.*, 2005. URL: http://eprints.ecs.soton.ac.uk/10862/01/wip2005.pdf.

[94] C Wroe, R Stevens, C Goble, A Roberts, and M Greenwood. A suite of DAML+OIL ontologies to describe bioinformatics web services and data. *International Journal of Cooperative Information Systems*, 12(2):197–224, 2003.

[95] Chris Wroe, Carole Goble, Mark Greenwood, Phillip Lord, Simon Miles, Juri Papay, Terry Payne, and Luc Moreau. Automating experiments using semantic data on a bioinformatics grid. *IEEE Intelligent Systems*, pages 48–55, 2004.

[96] XEMBL Web Service. URL: http://www.ebi.ac.uk/xembl.

[97] XScufl Language Reference. URL: http://www.ebi.ac.uk/ tmo/mygrid/XScuflSpecification.html.