An XML Mapping Language for Dynamic Semantic Workflow Harmonisation

Martin Szomszor [a,1], Terry R. Payne [c,*], Luc Moreau [b]

[a] *City University, Northampton Square, London EC1V 0HB, UK*
[b] *University of Southampton, Southampton, SO17 1BJ, UK*
[c] *University of Liverpool, Liverpool, L69 3BX, UK*

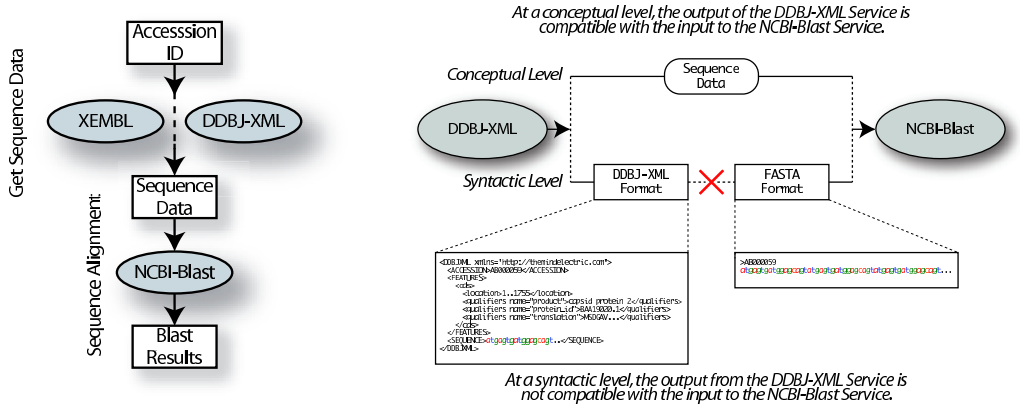
Abstract

Service-oriented architectures have evolved to support the composition and utilisation of heterogeneous resources, such as services and data repositories, whose deployment 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 middle-ware is required as 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 format incompatibilities are resolved, a step we refer to as syntactic mediation. Current solutions are entirely manual; users must consider the low-level (i.e. data format) 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 an 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 that facilitates mapping reuse. We present our mapping language (FXML-M) and give both its precise semantics, the rules that define the transformation process they dictate, and present an evaluation of an implementation of the approach with respect to other mapping mechanisms.

Key words: Semantic Web Services, XML Transformation, Lifting/Lowering, Grounding, Workflow Harmonisation, Syntactic Mediation

1 Introduction

The continuing uptake of the World Wide Web as a platform for the dissemination and sharing of products, services, and information, has pushed both the boundaries and uptake of distributed systems. Approaches such as Service Oriented Architectures, Multi-Agent Systems, and the Semantic Web, have evolved to support the growing complexity of interaction and co-ordination between parties. Much of the success of these computing paradigms can be attributed to their adoption of standard markup languages, such as XML, which are designed to support the mutually intelligible interchange of data. However, whilst this may support easier access and reuse of third-party services, they have failed to address many of the *knowledge-based problems* associated with the integration of disparate services, such as interface and data model heterogeneity. It has been argued [33,32] that while Web Service architectures have provided an effective infrastructure at the physical and syntactic level, they force application designers to make simplifying assumptions, such as the a-priori agreement of interface specifications and data models [36].



(a) A simple bioinformatics task: get sequence data from a database and perform a sequence alignment on it.

(b) The output from the DDBJ-XML Service is not compatible for input to the NCBI-Blast Service.

Fig. 1. Semantic similarity between services does not guarantee syntactic compatibility.

In an effort to increase interoperability, automation, and ease-of-use, recent research in the field has concentrated on aligning service description methodologies with Semantic Web ideologies [7], for example, through the use ontologies that capture the semantics of a service interface. Essentially, this annotation approach provides users with conceptual definitions of what a service does using domain specific terminology, supporting more intuitive service discovery and composition.

* Corresponding author. Tel: +44 (151) 795 4251
Email addresses: martin.szomszor.1@city.ac.uk (Martin Szomszor),
T.R.Payne@liverpool.ac.uk (Terry R. Payne),
l.moreau@ecs.soton.ac.uk (Luc Moreau).
[1] This research is funded in part by EPSRC myGrid project (reference GR/R67743/01)

With the introduction of semantically annotated Web Services [3,28], Workflow composition has shifted to a higher-level design process: users can choose to include services in a Workflow to achieve particular goals based on the conceptual service definitions. While this makes 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*. To harmonise these data incompatibilities, some form of data translation is required, often taking the form of a translation script, bespoke application, or the use of another Web Service or mediator [17,31,27]. In current systems, these *Type Adaptor* components must be discovered manually and inserted into the Workflow by hand, necessitating additional effort by the user. Thus, end users are distracted from the task of composing Workflows by the need to engineer and resolve incompatibility and interoperability issues between services.

Consider the following scenario. The MYGRID project provides an open-source Grid middle-ware that supports the construction of service workflows to support bioinformaticians. Using a service-oriented architecture, the MYGRID infrastructure provides a virtual workbench for supporting *in silico* biological experiments [14]. Access to data and computational resources is provided through Web Services, which can be composed using the workflow language XSCUFL [2] and executed with the FreeFluo [3] enactment engine. The biologist is provided with a user interface (Taverna [4]) which presents the services available, enables the biologist to compose and view workflows graphically, execute them, and browse the results.

A typical bioinformatics task may involve retrieving sequence data from a database and passing it to an alignment tool to check for similarities with other known sequences. Within MYGRID, this interaction is modelled as a simple Workflow, with each stage in the task being fulfilled by a Web Service, illustrated in Figure 1(a). Many Web Services are available for retrieving sequence data; the ones used in this example are DDBJ-XML (`http://xml.ddbj.nig.ac.jp/`) and XEMBL (`http://www.ebi.ac.uk/xembl/`) . To obtain a sequence data record, an accession number is passed as input to the service, and an XML document is returned. The documents returned from either service essentially contain the same information, namely the sequence data as a string (e.g. `atgagtga...`), references to relevant publications, and features of the sequence (such as the protein translation). However, the way this information is represented differs - XEMBL returns an EMBL [5] formatted record whereas DDBJ-XML returns a document using their own custom format. The next stage in the Workflow is to pass the sequence data

[2] http://www.ebi.ac.uk/~tmo/mygrid/XScuflSpecification.html
[3] http://freefluo.sourceforge.net/
[4] http://taverna.sourceforge.net/
[5] http://www.ebi.ac.uk/schema/EMBL_common.xsd

to an alignment service, such as the BLAST service at NCBI [6], which consumes a string of FASTA [7] formatted sequence data.

Intuitively, a bioinformatician will view the two sequence retrieval tasks as the same type of operation, expecting both to be compatible with the NCBI-Blast service. The semantic annotations used by the FETA service discovery engine [23] support this, as the output types are assigned the same conceptual type, namely a *Sequence_Data_Record* concept. However, when plugging the two services together, the resulting workflow fails due to data-format incompatibilities, as the output from either sequence data retrieval service is not directly compatible for input to the NCBI-Blast service (Figure 1(b)). To harmonise the Workflow, some intermediate processing or *syntactic mediation* is required to massage the data generated from the first service into a format suitable for input to the second service.

By exploiting the fact that semantically annotated Web Services can be connected through a shared conceptualisation, we present a scalable mediation approach in this paper that adopts these ontologies as an intermediate data model through which different syntactic representations may be converted. To support the specification of mappings between XML schemas and their corresponding ontology definitions, we present the declarative and composable XML mapping language, FXML-M, which includes a suitably rich set of operators to satisfy complex translation requirements derived from real bioinformatics data sources. A formalism is presented to give precise semantics for this mapping language, as well as rules to define the transformation of documents. An implementation of the rules that provide the semantics of this mapping language have been used as a basis for the creation of a *Configurable Mediator* - a dynamic Type Adaptor that converts instances of XML between different representations according to a set of composable mapping rules [35].

This paper is organised as follows: Section 2 gives an overview of our mediation approach and how OWL ontologies can be used to support data translation through an intermediate representation. In Section 3, we present our mapping language, FXML-M, utilising the bioinformatic data sources from the use case presented above to derive transformation requirements. Section 4 gives an overview of our Configurable Mediator with details of how an implementation of the mapping language (FXML-T) can be used to direct the data translation necessary to harmonise the use case Workflow; and is evaluated in Section 5. Section 6 contains related work before our conclusions are presented in Section 7.

[6] http://www.ncbi.nlm.nih.gov/BLAST/
[7] http://www.ebi.ac.uk/help/formats_frame.html

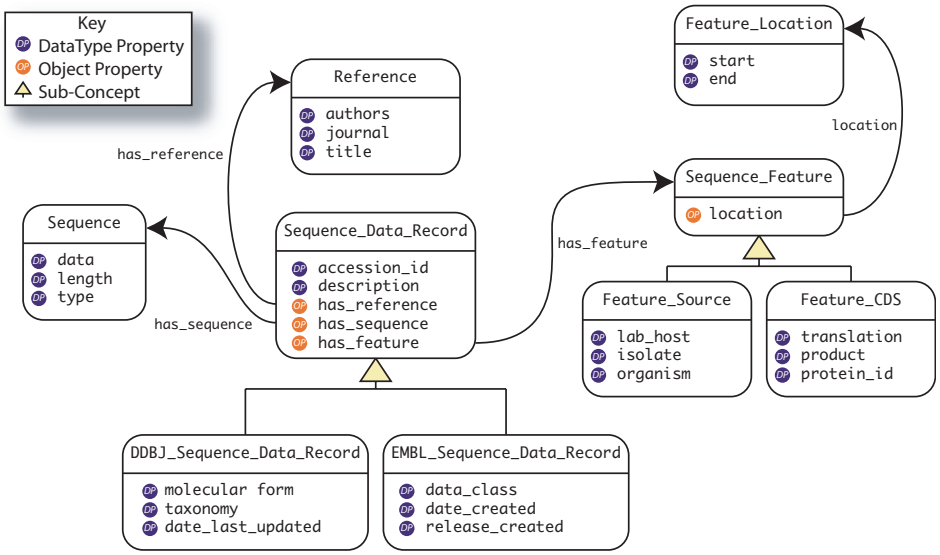


Fig. 2. An Ontology to describe sequence data records.

2 Using OWL for Scalable Syntactic Mediation

In large-scale open systems, such as the Grid and Web Services, service providers and consumers can assume arbitrary data formats to represent any information they consume and produce. As discussed above, this causes problems when services are connected in a Workflow based on their conceptual interface definitions.

Many systems employ a direct mediation approach: i.e. conversion components take the form of a translation script, bespoke program or Web Service [17,31,27]. However, as the number of compatible data formats increases, the number of Type Adaptors required is $O(n^2)$. Thus, we propose a modular and composable approach that uses an intermediate representation, based on a shared ontological conceptualisation. Using OWL to capture the structure and semantics of XML data has been used elsewhere [22,2] and has proven to be a useful for data integration. To illustrate this within our bioinformatics use case, Figure 2 presents an ontology to describe the Sequence Data Records presented earlier.

The main class, *Sequence_Data_Record* (centre of Figure 2), has the data-type properties *accession_id* (denoting the unique id of the data-set) 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*[8]. 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 other features, each having a *Feature_Location* that contains the *start* and *end* position of the feature in the sequence. As there are several different feature types, we show two of the more

[8] *Type* here does not denote a syntactic type, but rather the type, or *kind* of sequence.

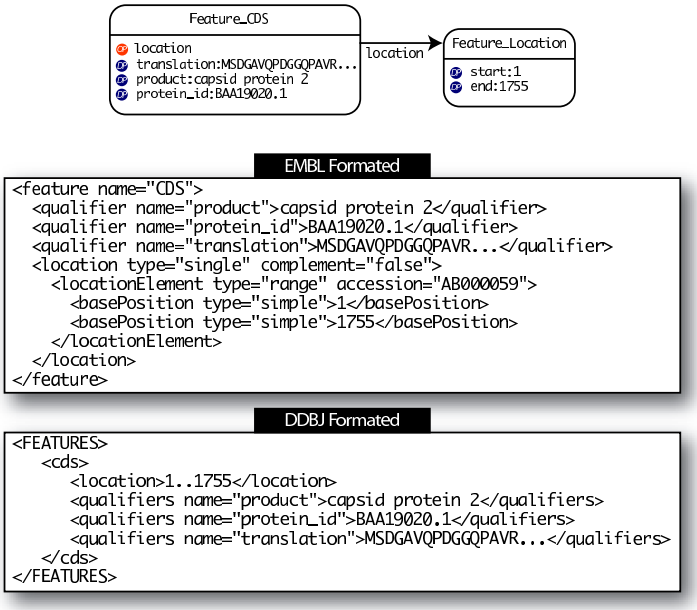


Fig. 3. EMBL and DDBJ formatted XML to describe the same sequence feature.

common ones in this example: *Feature_Source* and *Feature_CDS*; both of which are sub-classes of the *Sequence_Feature* concept. 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-classes: *DDBJ_Sequence_Data_Record* and *EMBL_Sequence_Data_record*. These classes capture the fact that while both the DDBJ-XML and XEMBL formats contain mainly the same information, they also include certain attributes that are unique to each format, e.g. repository specific information such as the date created or date last updated.

The XML fragments describing a sequence feature in both DDBJ-XML and EMBL formats are presented in Figure 3. These two representations contain the same information in different formats: the *Feature* is of type *CDS*, it has a *product*, *protein_id*, *translation* and *location*. This is illustrated at the top of Figure 3. An instance of the *Feature_CDS* class would be used with three data-type properties holding the *translation*, *produce* and *protein_id*. The feature location 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 domain ontology in place, syntactically incongruous data-flows between two services can be harmonised by translating data from one representation to another via the intermediate OWL model. This idea is illustrated in Figure 4 where the output from the DDBJ-XML service is converted to its corresponding concept instance (the *DDBJ_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*, i.e. the conversion of an XML document to an OWL concept instance; and *Con-*

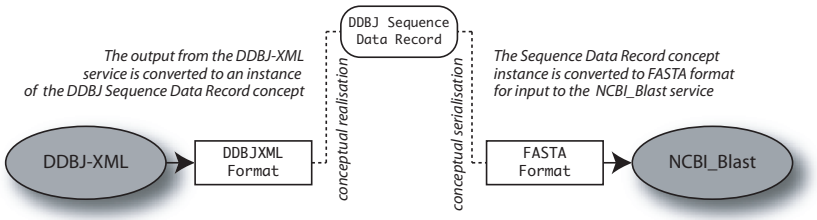


Fig. 4. Using an concept instance to harmonise services with incompatible interfaces.

ceptual Serialisation, i.e. the conversion of an OWL concept instance back to an XML document. These conversions are analogous to the *lifting* and *lowering* activities within the WSMO framework, whereby XML documents are *lifted* to WSML instances, and later *lowered* back to XML [24]. To simplify the specification of 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. Whilst OWL concept instances are typically specified using XML syntax, XML schemas do not usually exist to validate them. Therefore we automatically generate schemas using the OWL-$\mathcal{XIS}$ generator, presented below.

Klein et al. [21] present an algorithm to generate XML schemas that validate OIL [15] ontology representations. By extending this algorithm to cater for OWL ontologies, we can generate XML schemas to validate OWL concept instances for a given ontology definition. Our algorithm, on which our OWL-$\mathcal{XIS}$ (OWL XML instance schema) generator is based, is outlined as follows:

(1) Generate the entailed ontological model using a reasoner, such as Pellet [9] ;
(2) Create an XSD element for every OWL concept in the ontology;
(3) Create an XSD element for every OWL property in the ontology. For properties that link concepts to other concepts (i.e. an **object property**), such as the *has_sequence* property in our bioinformatics ontology, the type of the element is an XML complex type; whereas for properties that link concepts to literal values (i.e. a **data-type properties**), a predefined XSD type is used.
(4) Once the XSD elements have been created, an XML schema complex type is created for each concept. A list of all possible properties for the concept are extracted by checking the domain of all properties in the ontology. 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 one concept is a sub-concept of another, such as the *Feature_Source* concept, an XSD type extension is used to provide the inheritance of properties from the parent type.
(5) Finally, a type definition is created for every property in the ontology. When schema elements for object property types are created, the range of the property is examined and a list of possible concepts that property links to is determined. If an object property links to a concept which has sub concepts, such

[9] http://pellet.owldl.com/

as the *has_Feature* property, the complex type is set to be a choice over any of the sub concepts, e.g. the *has_Feature* complex type will be a choice of *Sequence_Feature*, *Feature_Source*, or *Sequence_CDS*.

When creating elements or complex types, the namespace and local name of the concept is mirrored in the XML schema; thus a URI pointing to a particular OWL concept or property also refers to the XML schema element that validates it. 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 itself exposed as a Web Service. The resulting XML schema can thus be used to support the definition of transitive mapping rules between different XML schematic descriptions. By identifying and composing the necessary mappings between the different XML data schema and their shared OWL concepts, dynamic mediation between services can be achieved. Before describing this harmonisation process (Section 4), we first present the mapping language formalism, FXML-M.

3 Mapping Language Formalism

To support the specification of mappings between different XML schemas, i.e. between those that describe concrete data formats such as the DDBJ-XML format, and those that describe valid concept instances from an ontology, we present the XML mapping and transformation formalism, FXML-M. In FXML-M, the mappings (from components [10] in a source XML schema to components in a destination XML schema) are used to define the transformation of an XML document from one representation to another. As the individual mappings provide transformations for XML components, libraries of such mappings can be maintained. This modular approach facilitates mapping reuse through the composition of new mapping bindings when XML schemas evolve, without the need for redefining complete mapping sets.

To illustrate the requirements for our mapping language, we show a subset of a full sequence data record in DDBJ-XML format and its corresponding OWL concept instance (serialised in XML according to the schema automatically generated by the OWL-$\mathcal{XIS}$ generator) in Figure 5. We consider six different mapping typesthat 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, 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),

[10] The term components is used to encompass elements, attributes, and literal values.

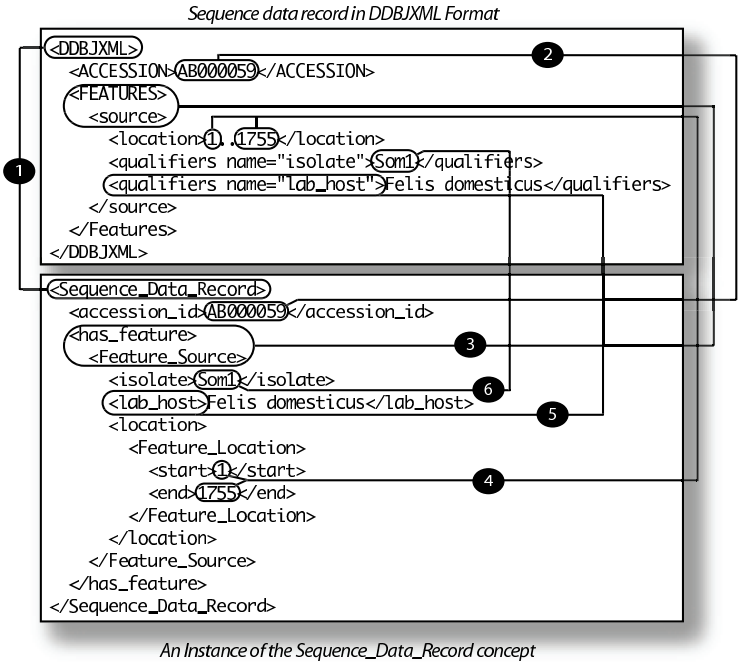


Fig. 5. Mappings between elements and attributes in the DDBJXML Sequence Data format and elements within the XML serialisation of the `Sequence_Data_Record` OWL concept.

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.

(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.

(4) **String manipulation support**
In complex cases, the contents of a string literal may contain two or more distinct pieces of information. In Figure 5, the `<location>` element has text containing the start and end position, delimited by "..". Each of these positions must be mapped to separate elements (i.e. `<start>` and `<end>`) 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 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

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://mygrid.org/schema"
 xmlns="http://mygrid.org/schema">

  <xsd:element name="a">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="b"
          type="xsd:string" minOccurs="1"
          maxOccurs="1"/>
        <xsd:element ref="c" minOccurs="1"
          maxOccurs="1"/>
      </xsd:all>
      <xsd:attribute name="id"
        type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="c" type="c-type"/>

  <xsd:complexType name="c-type">
    <xsd:sequence>
      <xsd:element ref="b" minOccurs="1"
        maxOccurs="2"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="c-extended">
    <xsd:complexContent>
      <xsd:extension base="c-type">
        <xsd:sequence>
          <xsd:element ref="d" minOccurs="1"
            maxOccurs="3">
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="b" type="xsd:integer"/>
  <xsd:element name="d" type="xsd:integer"/>

</xsd:schema>
```

Listing 1. A Simple XML Schema.

`"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-XML 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 facilitate this kind of behaviour, our mapping language provides local scoping to support the application of different rules in different contexts.

Because of these complex mapping requirements, we have specified our mapping language and the transformation of XML documents using a formalisation. This gives precise semantics for our language and helps us capture the more difficult

transformation properties such as predicate support and local scoping.

3.1 XML Formalisation

Our mapping and translation theory (presented below) is based on an existing XML and XML schema formalisation called Model Schema Language (MSL) [9]. While other XML formalisms have been proposed [6] [26], 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.

We first outline the principal features of MSL: how elements, attributes and types are referenced (Section 3.1.1), how groups of elements are specified for type declarations (Section 3.1.2), how XML schema components are defined (Section 3.1.3), and how XML documents are represented (Section 3.1.4). This should provide sufficient knowledge to understand our mapping and translation formalisation, presented in Section 3.2.

3.1.1 Normalised schema

MSL references the components of an XML schema (such as elements, attributes and types) using a normalised format. This normalisation assigns a unique, universal name to each schema part and provides a flat representation of the components found within a schema document, thus providing disambiguation between components with the same name that have been declared within different scopes. To illustrate this notation, we list (below) the normalised form for all the XML components declared in the simple XML schema shown in Listing 1, with corresponding line numbers in square brackets to show where they are defined. These references are used to point to XML schema components: the definition of the actual elements are presented later in Section 3.1.2.

```
[7] http://mygrid.org/schema/#element::a
[8] http://mygrid.org/schema/#element::a/type::*
[16] http://mygrid.org/schema
        /#element::a/type::*/attribute::id
[10] http://mygrid.org/schema/#element::a/type::*/element::b
[21] http://mygrid.org/schema/#element::c
[23] http://mygrid.org/schema/#type::c-type
[30] http://mygrid.org/schema/#type::c-extended
[41] http://mygrid.org/schema/#element::b
[42] http://mygrid.org/schema/#element::d
```

The first part of the normalised schema reference, delimited by 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) corresponding to the local name assigned to the component. For example, the element a is defined in the global scope (line 7 of Listing 1) and is referenced with the namespace prefix

http://mygrid.org/schema

and the normalised path reference element::a. The element acontains an anonymous complex type definition (line 8) 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 10) named bwhich can be distinguished from the globally defined element named b(line 41) because they have different normalised schema references (element::a/type::*/element::band element::brespectively). The type refinement given in line 30 is used later to illustrate type inheritance within MSL.

A short form notation is used throughout the rest of this paper 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 ausing a/*, and the attribute id (line 16) using a/*/@id.

3.1.2 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:stringor 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 the element a in our example above (Listing 1), 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 [1]. We let g range over model groups.

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

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

3.1.3 Components

In MSL, schema components (XML elements, attributes, etc...) can be one of seven sorts [11] : *element*, *attribute*, *simply type*, *complex type*, *attribute group* or *model group*. We let srt range over sorts, where sort $srt \in \{$ `attribute`, `element`, `simpleType`, `complexType`, `attributeGroup`, `modelGroup` $\}$

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 1. 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 component name, b is a boolean value and g is a model group. 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:

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

[7]
component(
  sort = **element**,
  name = **a**,
  base = **xsd:UrElement**,
  derevation = **restriction**,
  refinement = {},
  abstract = **false**,
  content = **a/***
)

[8]
component(
  sort = **complexType**,
  name = **a/***,
  base = **xsd:UrType**,
  derevation = **restriction**,
  refinement = {**restriction, extension**},
  abstract = **false**,
  content = **a/*/@id{1,1} & a/*/b{1,1} & c{1,1}**
)

[10]
component(
  sort = **element**,
  name = **a/*/b**,
  base = **xsd:UrElement**,
  derevation = **restriction**,
  refinement = {},
  abstract = **false**,
  content = **xsd:string**
)

[16]
component(
  sort = **element**,
  name = **a/*/@id**,
  base = **xsd:UrAttribute**,
  derevation = **restriction**,
  refinement = {**restriction**},
  abstract = **false**,
  content = **xsd:string**
)

[21]
component(
  sort = **element**,
  name = **c**,
  base = **xsd:UrElement**,
  derevation = **restriction**,
  refinement = {},
  abstract = **false**,
  content = **c-type**
)

[23]
component(
  sort = **complexType**,
  name = **c-type**,
  base = **xsd:UrType**,
  derevation = **restriction**,
  refinement = {**restriction, extension**},
  abstract = **false**,
  content = **b{1,2}**
)

[30]
component(
  sort = **complexType**,
  name = **c-extended**,
  base = **c-type**,
  derevation = **extension**,
  refinement = {**restriction, extension**},
  abstract = **false**,
  content = **d{1,3}**
)

[41]
component(
  sort = **element**,
  name = **b**,
  base = **xsd:UrElement**,
  derevation = **restriction**,
  refinement = {},
  abstract = **false**,
  content = **xsd:integer**
)

[42]
component(
  sort = **element**,
  name = **d**,
  base = **xsd:UrElement**,
  derevation = **restriction**,
  refinement = {},
  abstract = **false**,
  content = **xsd:integer**
)

Fig. 6. MSL to represent the schema components defined Listing 1 with listing line numbers for components indicated in square brackets.

$$
\begin{aligned}
cmp ::= & \texttt{component (} \\
& \quad \texttt{sort =}\ srt \\
& \quad \texttt{name =}\ x \\
& \quad \texttt{base =}\ x \\
& \quad \texttt{derivation =}\ der \\
& \quad \texttt{refinement =}\ ders \\
& \quad \texttt{abstract =}\ b \\
& \quad \texttt{content =}\ g \\
& \texttt{)} \\
der ::= & \texttt{extension} \mid \texttt{refinement} \\
ders ::= & \{der_1, \ldots, der_l\}
\end{aligned}
$$

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 1) can be defined as in Figure 6 (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).

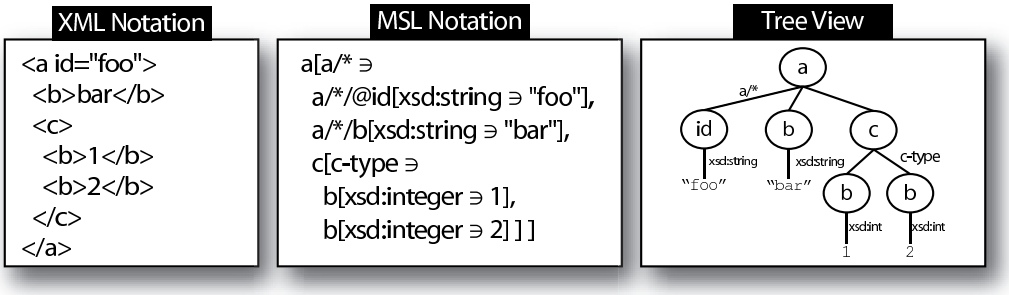


Fig. 7. XML notation, MSL notation, and a tree view of a simple XML document.

3.1.4 Typed Documents

In the previous Sections (3.1.1, 3.1.2 and 3.1.3), we 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:

$td ::=$	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

To illustrate this, Figure 7 contains XML and MSL notation to represent the same XML document adhering to the schema presented earlier in Listing 1. A tree view is also provided to visualise the data sample. 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`.

3.2 Formalisation Extensions

Before describing our XML mapping and transformation methodology, we present two extensions to the MSL formalisation: the 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.

3.2.1 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 [12] . 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. 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.

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 paper, we use inference rule notation [13]. 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 8. Rule `PATHC.A` states that a path component θ referencing an attribute *a* matches the typed document a[t $\ni$ td_c], and therefore $\theta \vdash td_s \rightarrow$ a[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 element 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 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:*

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

[12] http://www.w3.org/TR/xpath

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Rules to define the application of path components to typed documents.

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

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

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

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

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

Fig. 9. Rules to define the direct children of typed documents

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. The inference rules used to describe this behaviour are given in Figure 9. Rule `CHILD.A` states that a typed document td_s that is the attribute definition $\text{a}[\text{t} \ni td_c]$ contains the document td_c. A similar definition is used to define the contents of an element in rule `CHILD.E`. The other three rule define the contents of the empty document (`CHILD.EMP`), a constant (`CHILD.C`), and a sequence of typed documents (`CHILD.SEQ`) to be itself.

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

Fig. 10. A rule to define the application of a path expression to a typed document.

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:*

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

Given these rules that describe the contents of typed documents and the matching of path components, the evaluation of a path expression can be specified (as in Figure

10). The resulting document, td_n, is taken from the contents of the final component matched ($\mathbf{child}(td_{n-1'}) = td_n$).

To illustrate by means of an example, the path expression $\Theta = \langle \mathtt{a}, \mathtt{a}/*/@\mathtt{id}, value \rangle$ can be evaluated against the typed document given in Figure 7 to give the result `"foo"`, and would be equivalent to applying the XPATH statement `a/@id/text()`. To illustrate this evaluation, Figure 11 shows the steps involved, and an explanation of the rules used is given below:

(1) The source document is td_s and the path expression is Θ. Rather than write the full typed document, the string "... " 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 ($\mathtt{a} \vdash td_s \rightarrow \mathtt{a}[\mathtt{a}/* \ni \ldots]$) using the rule `PATHC.E`.
(3) $[\alpha']$ - The direct child of the matched document is found using the 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.

3.2.2 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 3 (example mapping 5) where the `<qualifiers>` element is mapped differently depending on the value of the `@name` attribute. The predicate atom $patom$ ranges over path expressions and constants (such as a string or a number):

$$\begin{array}{rcll} patom ::= & \Theta & & \text{path expressions} \\ \mid & c & & \text{constant} \end{array}$$

A predicate ψ is then defined (see Figure 12).

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

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

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

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

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

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

$$
\dfrac{td_s = \texttt{foo}}{\textbf{child}(\texttt{foo}) = \texttt{foo}} \ (\texttt{CHILD.C}) \quad [\gamma']
$$

Fig. 11. An example path expression evaluation to retrieve the contents of an attribute.

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

Fig. 12. The definition of predicate ψ.

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

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

Fig. 13. Rules to define the evaluation of predicate expressions.

Definition 4 (Predicate Evaluation) *Predicates can be used for determining the 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:*

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

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 13) 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 $\Theta \vdash td_s \rightarrow td_r$ (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 for defining the evaluation of predicates are given in Figure 14. 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 ele-

ments 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.

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

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

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

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

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

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

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

Fig. 14. Rules to define the evaluation of predicates.

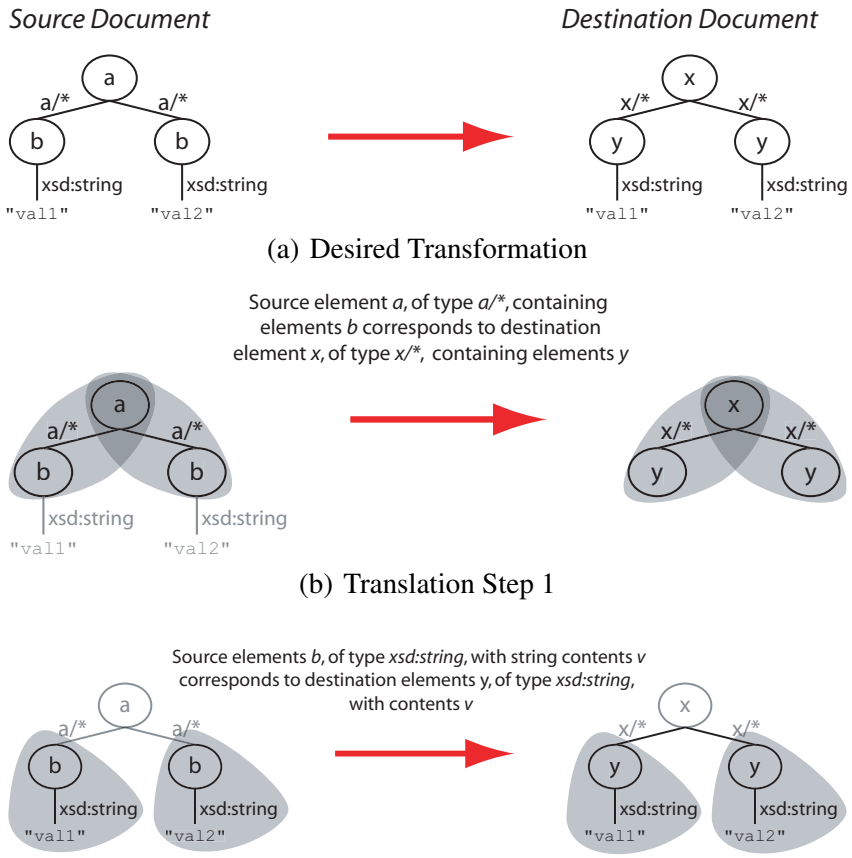


Fig. 15. Transformation through recursion.

3.3 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, as illustrated in Figure 7. By viewing an XML document as a tree, we can visualise the transformation process using an 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 illustrated in Figure 15 using a trivial transformation. Using this method, we can describe a translation using a number of *mappings* that 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 3.4 where we also describe more complex mapping constructs.

3.4 Mappings and the Transformation Process

In this sub-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.

3.4.1 Mapping Paths

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

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

Definition 5 (Source Mapping Pairs) *Each pair in a source mapping path contains a path component (θ) that matches* XML *components from the source 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 that 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 16 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`.

A joining operator is used to define the creation of components in the destination document. We let ω range over joining operators:

$$\begin{array}{rl} \text{joining operator } \omega ::= & join \\ \mid & branch \end{array}$$

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

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

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

Fig. 16. Rules to define the evaluation of source mapping paths.

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 3.4.2, together with the joining operator.

3.4.2 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:

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

where 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. As the order in which the mappings are defined corresponds to the order in which they are applied, a binding is therefore defined as a sequence.

3.4.3 Transformation

The application of a Binding to a typed source 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 the set of applicable mappings, M_a, from B which are applicable to td_s.

(2) **Source Document Selection**
Given M_a, and a source document td_s, for each mapping $m_x \in M_a$ the source mapping path ρ from m_x to generate 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: a) no mappings are valid; b) the empty document is encountered; or c) 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.

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

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

Fig. 17. Rules to define mapping compatibility.

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

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

The rule `COMP.ME` in Figure 17 states that when the first component referenced in a source mapping path is the element e, and the source document td also corresponds to 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 de-referencing map that takes a component name x and gives the corresponding component so that features of the component (such as its type) can be determined:

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

The most complex stage in the translation process is the construction of the destination typed document. This stage is complex due to the necessity of creating multiple

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

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

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

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

Fig. 18. Rules to define the construction of destination documents (base case).

elements in order to map components from the source domain to multiple components in the destination schema. We illustrate this problem in Figure 19, where a simple source document is translated into 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 $\rho \vdash td_s \rightarrow td_c$ from the 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. To denote the construction of the destination document, we write:*

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

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 18. 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); the resultant pairs form a destination creation set $R = \{P_1, P_2, \ \ldots \ , P_n\}$. 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 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 20 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 20 defines the path component x located in the set of joined destination creation pairs R_{join}.

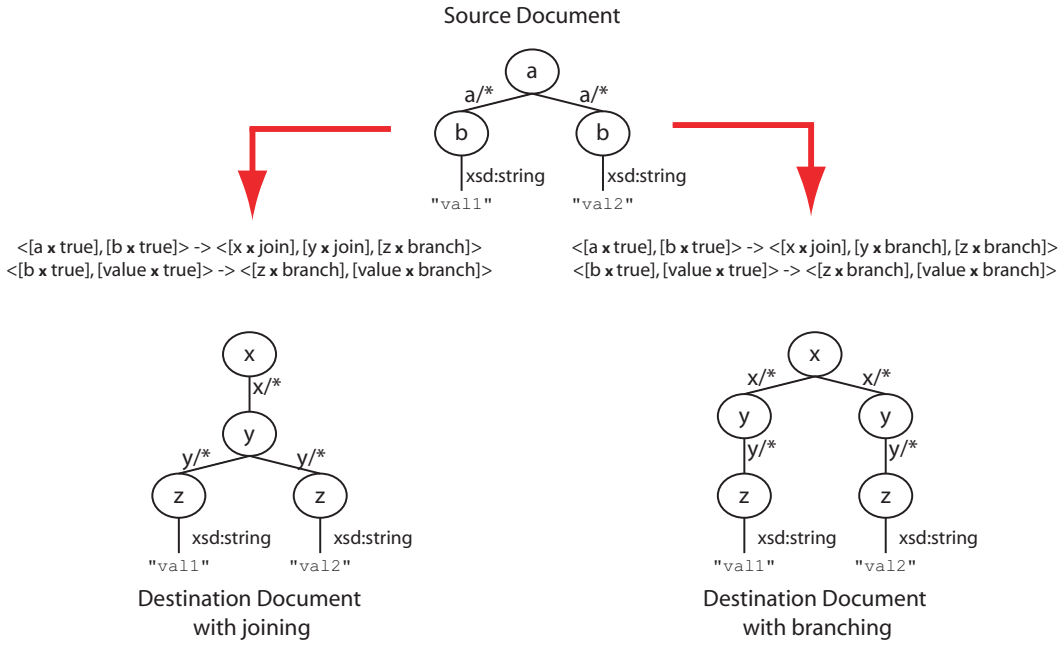


Fig. 19. A Source Document with two possible transformations, each using a different joining operator.

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

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

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

Fig. 20. Rules to define the sets of joined and branched destination creation pairs.

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

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

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

Fig. 21. Rules to define the construction of sequences.

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

$$td_a \sqcap td_b = td_r$$

Figure 21 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 empty 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, R can be used to construct the destination document td_r. To denote this we write:*

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

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

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

$$\texttt{RBRANCH.EVAL}\quad \frac{R_{branch} = \{P_1, P_2, \ \ldots\ , P_k\} \quad \mathbf{construct}(P_1) = td_1, \quad \ldots\ , \quad \mathbf{construct}(P_n) = td_n}{\mathbf{construct}(R_{branch}) = td_1 \sqcap \ \ldots\ \sqcap td_n}$$

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

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

Fig. 22. Rules to define the construction of the destination document.

Figure 22 contains rules to define the construction of documents using the set of destination creation pairs R. Rule `R.EVAL` states the 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 21).

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

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

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

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

Fig. 23. Rules to define the evaluation of Bindings.

removed to give P_i' ($\mathbf{next}(P_i) = P_i'$ using rule NEXT.C.PAIR). These new destination content pairs are combined in the set R' which is 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 18) 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

$m_1 = \langle$ $\langle$DDBJXML, ACCESSION$\rangle$, $\langle$[Sequence_Data_Record $\times$ $join$], [accession_id $\times$ $branch$]$\rangle$, $\emptyset\rangle$

$m_2 = \langle$ $\langle$ACCESSION, *value*$\rangle$, $\langle$[accession_id $\times$ $join$], *value*$\rangle$, $\emptyset\rangle$

$m_3 = \langle$ $\langle$DDBJXML, DEFINITION$\rangle$, $\langle$[Sequence_Data_Record $\times$ $join$], [definition $\times$ $branch$]$\rangle$, $\emptyset\rangle$

$m_4 = \langle$ $\langle$DEFINITION, *value*$\rangle$, $\langle$[definition $\times$ $join$], *value*$\rangle$, $\emptyset\rangle$

$m_7 = \langle$ $\langle$source, location$\rangle$, $\langle$[Feature_Source $\times$ $join$], [has_position $\times$ $branch$], [Location $\times$ $branch$]$\rangle$, $\emptyset\rangle$

$m_9 = \langle$ $\langle$location, *value*{"^[^.]+"}$\rangle$, $\langle$[Location $\times$ $join$], [start $\times$ $branch$], *value*$\rangle$, $\emptyset\rangle$

$m_{10} = \langle$ $\langle$location, *value*{"[^.]+"}$\rangle$, $\langle$Location $\times$ $join$], [end $\times$ $branch$], *value*$\rangle$, $\emptyset\rangle$

$m_{11} = \langle$ $\langle$DDBJXML, FEATURES, source$\rangle$,
$\langle$[Sequence_Data_Record $\times$ $join$], [has_feature $\times$ $branch$], [Feature_Source $\times$ $branch$]$\rangle$, $\emptyset\rangle$

$m_{12} = \langle$ $\langle$source, [qualifiers $\times$ {qualifiers, qualifiers/*/@name*value* = "isolate"}]$\rangle$,
$\langle$[Feature_Source $\times$ $join$], [isolate $\times$ $branch$]$\rangle$, $(m_{13})\rangle$

$m_{13} = \langle$ $\langle$qualifiers, *value*$\rangle$, $\langle$[isolate $\times$ $join$], *value*$\rangle$, $\emptyset\rangle$

$m_{14} = \langle$ $\langle$source, [qualifiers $\times$ {qualifiers, qualifiers/*/@name*value* = "lab_host"}]$\rangle$,
$\langle$[Feature_Source $\times$ $join$], [lab_host $\times$ $branch$}]$\rangle$, $(m_{15})\rangle$

$m_{15} = \langle$ $\langle$qualifiers, *value*$\rangle$, $\langle$[lab_host $\times$ $join$], *value*$\rangle$, $\emptyset\rangle$

Fig. 24. Example mapping to convert and instance of a DDBJ-XML formatted sequence data record to an OWL concept instance.

pair P has its first destination mapping pair removed to give P' ($\mathbf{next}(P) = P'$). 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 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 23. 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 $R = \{P_1, P_2, \ \ldots \ , P_n\}$.

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:*

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

The rule BINDING.EVAL presented in Figure 23 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.

3.5 Example Mappings and XML Syntax

To demonstrate our mapping language, we provide a subset of mappings (Figure 24) that transform an instance of a DDBJ-XML sequence data record to a *Sequence_Data_Record* concept instance. In this example, assume all source mapping path predicates are $true$ unless otherwise specified (see mapping 12 and 14). They are then used to define a binding B as follows:

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

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-XML format can then be evaluated using this binding to give a destination document which is the sequence data record in its corresponding OWL representation.

The specification of these mappings and the binding B is represented using XML in Listing 2. This XML document represents an M-Binding and can be used to drive the translation of XML documents, as we show below in Section 4. Our XML binding format is designed to look similar to conventional XPATH notation so users familiar with XML tools will find it intuitive. 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).

4 Implementation - The Configurable Mediator

The Configurable Mediator (C-MEDIATOR) is a software component that consumes M-Binding documents (such as that given in Listing 2), and uses them to

```
<binding name="DDBJ-to-sequencedata"
         xmlns="http://mygrid.org/schema/binding"
         xmlns:sns="http://mygrid.org/schema/DDBJ"
         xmlns:dns="http://mygrid.org/ont/sequencedata"
         targetNamespace="http://mygrid.org/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 2. An XML representation of the Binding.

direct the transformation of XML data from one format to another via the corresponding intermediate OWL representation [32]. The FXML-M formalism presented above was implemented as a SCHEME [19] library called FXML-T (Formalised XML Translation) to construct a *Translation Engine* which is combined with the JENA ontology processing API to create the Configurable Mediator. As discussed above, the transformation process consists of three stages: (i) conversion from the source

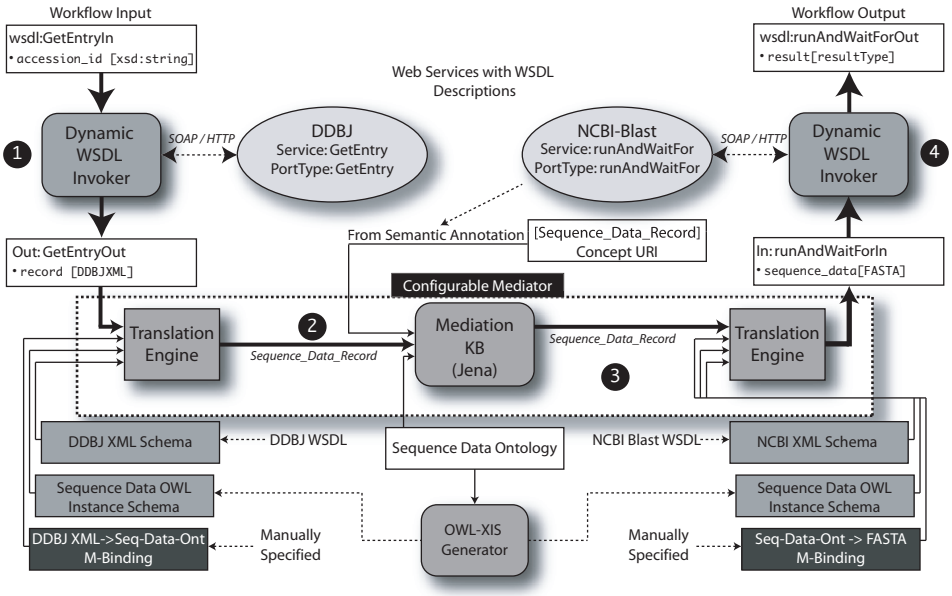


Fig. 25. A detailed view of the Configurable Mediator in the context of our use case.

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* [13], which provides an implementation of the transformation rules presented in Section 3.

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 DDBJ-XML service (an instance of the *DDBJ_Sequence_Data_Record* concept) is also classified as an instance of the *Sequence_Data_Record* concept. Therefore, input to a service consuming *Sequence_Data_Record*, such as the NCBI-Blast service, is valid. The C-MEDIATOR and its interaction with our Dynamic Web Service Invoker - DWSI [14] and the two target Web Services from our use-case is illustrated in Figure 25, which demonstrates how the C-MEDIATOR converts data from DDBJ-XML format to FASTA format via an instance of the *Sequence_Data_Record* concept. The figure also illustrates all the necessary documents 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

[13] Full details of the transformation engine and an evaluation of its performance can be found in [35].
[14] Full details of the DWSI and its evaluation can be found in [34].

in Figure 25:

(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 DDBJ-XML.
(2) The DDBJ-XML 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 into the 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.

5 Evaluation

To evaluate our SCHEME based FXML-T implementation of the FXML-M formalism, as well as the scalability of the language design itself, we devised several experiments to examine the performance of our Translation Engine against increasing document sizes, increasing schema sizes, and M-Binding compositions of increasing complexity. All the experiments were carried out using a 2.6 Ghz Pentium 4 PC with 1GB RAM running Linux (kernel 2.6.15-20-386) using unix utility `time`to record program user times. FXML-T was executed using the Guile Scheme Interpreter v1.6 [15] . Results are averaged over 30 runs so plotted values are statistically significant at a 95% confidence interval.

The scalability of FXML-T was investigated by increasing input document size (while maintaining uniform input XML schema size), and by increasing both input schema size and input document size. Our hypothesis stated that *expanding document and schema size would increase the translation cost linearly*. For comparison, FXML-T is tested against the following XML translation tools:

[15] `http://www.gnu.org/software/guile/guile.html`

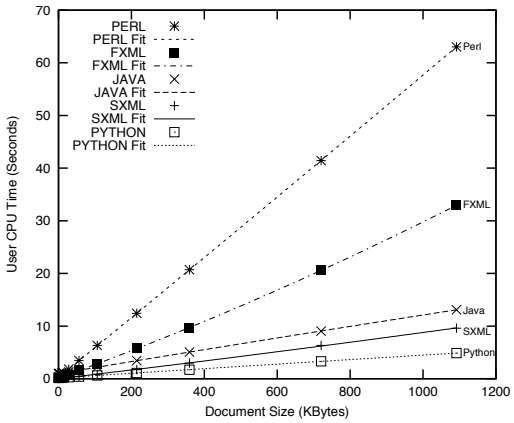


Fig. 26. Transformation Performance against increasing XML document size

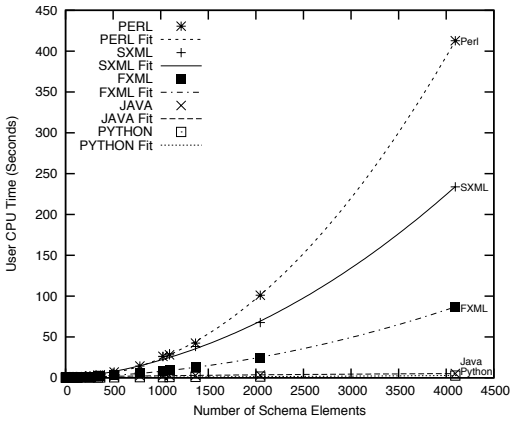


Fig. 27. Transformation Performance against increasing XML schema size

- XSLT: Using Perl and the XML::XSLT module [16] .
- XSLT: Using JAVA (1.5.0) and Xalan (v2.7.0) [17] .
- XSLT: Using Python (v2.4) and the 4Suite Module (v0.4) [18] .
- SXML: A SCHEME implementation for XML parsing and conversion (v3.0) [19] .

Since FXML-T is implemented using an interpreted language, and Perl is also interpreted, one would expect them to perform slowly in comparison to JAVA and Python XSLT which are compiled [20] . Figure 26 shows the time taken to transform 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 26 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.

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 176 Kbytes, while using 4095 elements a source document is 378 Kbytes. Figure 27 shows translation time against the number of schema elements used.

[16] `http://xmlxslt.sourceforge.net/`
[17] `http://xml.apache.org/xalan-j/`
[18] `http://4suite.org/`
[19] `http://okmij.org/ftp/Scheme/SXML.html`
[20] Although Python is interpreted, the 4Suite library is statically linked to natively compiled code

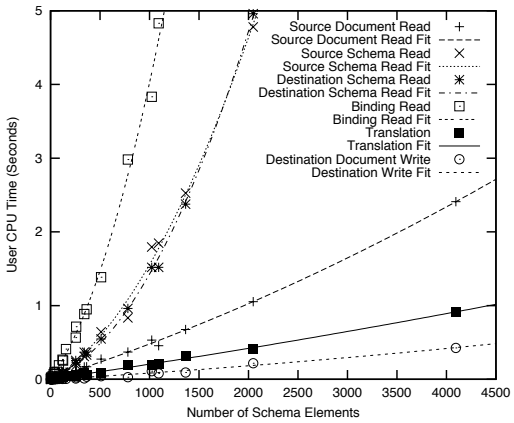


Fig. 28. FXML-T transformation performance breakdown against increasing XML schema size

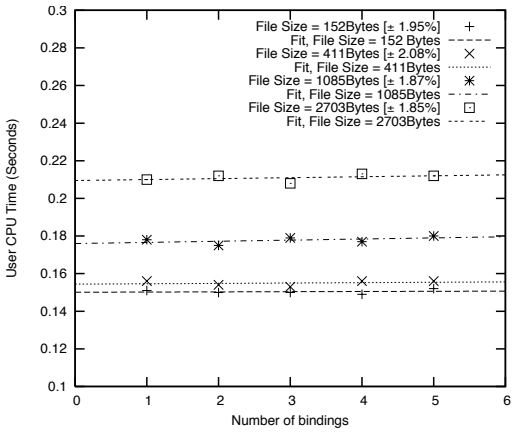


Fig. 29. Transformation Performance against number of bindings

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 28), 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 28). The SCHEMEXML 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.

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 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 the 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 run each test using 4 source documents with sizes 152 Bytes, 411 Bytes, 1085 Bytes, and 2703 Bytes. 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 29 shows the time taken to

transform the same 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.

6 Related Work

The idea of assigning semantics (or meaning) to elements and attributes inside XML schemas has been explored in a variety of 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 [20] 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 [30] employed a set theory approach to assign domain semantics to information represented in XML. A three-layer meta-model graph decomposed XML into three levels: (i) the instance-level graph models elements, attributes, and literals as nodes of a graph, and types as their edges; (ii) the type-level graph models an 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. As instances of elements in different schemas that share the same meta-type-level nodes are conceptually equivalent within this model, a homogeneous view for querying XML data across different logical representations (i.e. different XML schemas) can be achieved through the meta-type level.

Mrissa *et al* [25] utilised a local context ontology for augmenting ontological concepts to support ontological heterogeneities between Semantic Web Services. Service providers could attach additional metadata to an OWL concept definition to define extensional information (for example, a price concept may include additional details regarding currency, tax, scalar quantities, and data format). This metadata is then used to construct (on the fly) mediator services that perform the translation of data between services. In addition, a mechanism for inserting the description of these mediators into a BPEL4WS composition was also developed.

Liu *et al* [22] presented 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 be-

tween 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* [2] who define a mapping language to express the relationship between XML DTDs and OWL ontologies. Other efforts [5,12] also propose similar approaches to map XML schemas to RDF or OWL. However, in our approach we assume that both OWL ontologies, and the XML schemas already exist.

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 was explored by Balzer and Liebig [4] 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.

Hull et al [17,16] investigated the workow harmonisation problem within the MYGRID project. They argue that conversion services, or shims, can be placed in between services whenever some form of translation is required. Such shim services are *experimentally neutral*; i.e. they have no effect on the result of the experiment other than to change the representation of the data in some way. By enumerating the types of shims required in bioinformatics Grids and classifying all instances of shim services, they propose that the necessary translation components could be automatically inserted into a workow at run-time, or suggested to the user through an editor or composition tool at workflow design time. The shims proposed encapsulate a variety of conversion services, not just those that perform syntactic mediation. Whilst their *syntax translator* shim addresses a similar problem to that addressed in the paper, such shims were bespoke services that provided a 1-to-1 mapping between different XML schemas, and thus could not be dynamically composed and adapted for new scenarios at run-time.

The Piazza query-answering system [18] modelled variations at the data-level between the ontologies or schemas assumed by different systems. As values at this level may have a variety of different representations, as illustrated earlier in Fig 5, *concordance tables* are often used to map the associations between the values in the corresponding representations. Whilst a number of methods for mapping between ontologies have been proposed [10], mapping at the data-level can often require

some form of *structural* transformation, which may consist of decomposing, or aggregating a number of different data elements. The Piazza system proposed a structural transformation approach that exploited elements from XQUERY to construct directional mapping definitions which could be composed together to form a complete mapping between two schemas. Each mapping consisted of an XML template that maps some path or sub-tree of a legal instance of a target schema fragment, with embedded XQUERY statements that bind variables to XML nodes. These variables could then be referenced anywhere within the scope of the tagged fragment. Whilst Piazza mappings were primarily intended to map between XML schemas, they could also be used to convert arbitrary XML into RDF.

The SEEK project [8] specifically addressed 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 applicable to the services within the bespoke SEEK framework.

The Web Services Modelling Ontology (WSMO) [29] built upon, and extending the earlier UPML [11] framework. 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. Components within the WSMO architecture communicate using a standardised message format: an XML serialisation of the WSML language; thus, all participants within a WSMO framework are expected to communicate at a conceptual level using XML serialisations of WSML concepts through a process of *lifting* and *lowering*. To accommodate differences in conceptual representation, the WSMO infrastructure also contains explicit mediator components that support the translation of information between different WSML representations. Message adaptors are placed in-front of services 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. 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).

7 Conclusions and Future Work

In this paper, we illustrate the problem that arises when syntactically incompatible service interfaces are joined within a Semantic Web Services Workflow, through the use of a bioinformatics Grid scenario. By using OWL as an intermediate representation to capture the structure and semantics of differently formatted data-sets and interfaces that represent conceptually equivalent information, we have proposed a scalable mediation approach that facilitates the introduction of new formats, and minimises the number of *Type Adaptors* required to achieve maximum interoperability. This approach has been evaluated through an implementation, and compared with similar, hand-crafted mappings to verify the validity and feasibility of our approach.

The main contribution of this paper is the mapping language FXML-M, designed specifically to cater for the specification of mappings between XML and OWL in a modular and composable fashion. By using bioinformatics data sets from our use-case, we have been able to derive a complex set of mapping and transformation requirements. To this end, FXML has been created with the following novel language constructs:

- **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.
- **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-XML sequence data record, predicate evaluation is used. 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}) in Section 3.5.
- **String Manipulation**: When the value of an element contains two distinct entities, such as the `<location>`element in the DDBJ-XML 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 FXML-M mapping language has been realised through an implementation of the Configurable Mediator (C-MEDIATOR), which was briefly introduced in Section 4. When coupled with the Dynamic Web Service Invoker - DWSI [34], the FXML-M language provides a declarative mechanism for describing mappings that, once specified, can be automatically composed and executed at runtime to provide data harmonisation. An evaluation of the FXML-M language, as well as details of the implementation of our C-MEDIATOR and a comparison with other approaches is presented in [35].

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>`. Since FXML-M covers a large number of constructs from XPATH, future work could also use FXML-M as a basis to formalise XSLT and XQUERY.

References

[1] A. V. Aho, Algorithms for finding patterns in strings, in: Handbook of Theoretical Computer Science, MIT Press / Elsevier, Cambridge, Massachusetts, 1990.

[2] Y. An, A. Borgida, J. Mylopoulos, Inferring complex semantic mappings between relational tables and ontologies from simple correspondences., in: Int. Conf. on Ontologies, Databases and Applications of Semantics (ODBASE), Springer Verlag, Berlin Heidelberg, Germany, 2005, pp. 1152–1169.

[3] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, K. Sycara, DAML-S: Web service description for the semantic web, in: Proceedings of the 1st International Semantic Web Conf. (ISWC 02).

[4] S. Balzer, T. 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, Hiroshima, Japan, 2004, pp. 16–30.

[5] S. Battle, Gloze: Xml to rdf and back again, in: In Proceedings of the First Jena User Conference, 2006.

[6] D. Beech, A. Malhotra, M. Rys, A formal data model and algebra for xml, w3C XML Query Working Group Note (1999).

[7] T. Berners-Lee, J. Hendler, O. Lassila, The semantic web, Scientific American 284 (5) (2001) 34 – 43.

[8] S. Bowers, B. Ludascher, An ontology-driven framework for data transformation in scientific workflows, in: Intl. Workshop on Data Integration in the Life Sciences (DILS'04).

[9] A. Brown, M. Fuchs, J. Robie, P. Wadler, MSL - a model for W3C XML Schema., in: WWW, ACM, New York, NY, USA, 2001, pp. 191–200.

[10] J. Euzenat, P. Shvaiko, Ontology Matching, Springer-Verlag, 2007.

[11] D. Fensel, R. Benjamins, E. Motta, B. Wielinga, UPML: A framework for knowledge system reuse, in: Proceedings of the International Joint Conference on AI (IJCAI-99), Stockholm, Sweden, 1999.

[12] M. Ferdinand, C. Zirpins, D. Trastour, Lifting xml-schema to owl, in: Fourth International Conference on Web Engineering (ICWE), Springer-Verlag Berlin Heidelberg, 2004, pp. 354–358.

[13] G. Frege, Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought., in: From Frege to Godel: A Source Book in Mathematical Logic, 1879 - 1931, Harvard University Press, Cambridge, Massachusetts, 1967.

[14] C. Goble, S. Pettifer, R. Stevens, C. Greenhalgh, Knowledge Integration: In silico Experiments in Bioinformatics, in: I. Foster, C. Kesselman (eds.), The Grid: Blueprint for a New Computing Infrastructure Second Edition, Morgan Kaufmann, 2003.

[15] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, E. Motta, OIL: The Ontology Inference Layer, Tech. Rep. IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences, see http://www.ontoknowledge.org/oil/ (Sep.).

[16] D. Hull, R. Stevens, P. Lord, Describing web services for user-oriented retrieval, in: W3C Workshop on Frameworks for Semantics in Web Services, Digital Enterprise Research Institute, (DERI), Innsbruck, Austria, 2005.

[17] D. Hull, R. Stevens, P. Lord, C. Wroe, C. Goble, Treating shimantic web syndrome with ontologies, in: First AKT workshop on Semantic Web Services (AKT-SWS04) KMi, The Open University, Milton Keynes, UK. December 8, 2004, 2004, workshop proceedings CEUR-WS.org ISSN:1613-0073.

[18] Z. G. Ives, A. Y. Halevy, P. Mork, I. Tatarinov, Piazza: mediation and integration infrastructure for semantic web data, J. Web Sem. 1 (2) (2004) 155–175.

[19] R. Kesley, W. Clinger, J. Rees, Revised (5) report on the alogrithmic language scheme, Higher-Order and Symbolic Computation (1998) 7 – 105.

[20] H. H. Kim, S.-S. Park, Semantic integration of heterogeneous xml data sources, in: OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems, Springer-Verlag, London, UK, 2002, pp. 95–107.

[21] M. Klein, D. Fensel, F. van Harmelen, I. 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, 2000.

[22] S. Liu, J. Mei, A. Yue, Z. Lin, XSDL: Making xml semantics explicit, in: Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB2004), Springer Verlag, Berlin Heidelberg, Germany, 2005, pp. 64–83.

[23] P. Lord, P. Alper, C. Wroe, C. Goble, Feta: A light-weight architecture for user oriented semantic service discovery, in: Proc. of the 2nd European Semantic Web Conference, ESWC 2005, Springer Verlag, Berlin Heidelberg, Germany, 2005, pp. 17 – 31.

[24] M. Moran, A. Mocan, Towards translating between xml and wsml based on mappings between xml schema and an equivalent wsmo ontology, in: 2nd WSMO Implementation Workshop (WIW 2005), 2005.

[25] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg, S. Dustdar, A context-based mediation approach to compose semantic web services, ACM Trans. Internet Technol. 8 (1) (2007) 4.

[26] M. Murata, D. Lee, M. Mani, K. Kawaguchi, Taxonomy of xml schema languages using formal language theory, ACM Trans. Inter. Tech. 5 (4) (2005) 660–704.

[27] M. Paolucci, N. Srinivasan, K. Sycara, Expressing WSMO mediators in OWL-S, in: Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Springer Verlag, Berlin Heidelberg, Germany.

[28] D. Roman, U. Keller, H. Lausen, R. L. Jos de Bruijn, M. Stollberg, A. Polleres, C. Feier, C. Bussler, D. Fensel, Web service modeling ontology, Applied Ontology 1 (1) (2005) 77–106.

[29] D. Roman, H. Lausen, U. Keller, D2v1.0. web service modeling ontology (WSMO), WSMO Working Draft (September 2004).

[30] W. Schuetzelhofer, 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, IEEE Computer Society, Washington, DC, USA, 2002, p. 120.

[31] M. Stollberg, E. Cimpian, A. Mocan, D. Fensel, A semantic web mediation architecture, in: M. T. Kone, D. Lemire (eds.), The Canadian Semantic Web Working Symposium (CSWWS), vol. 2 of Semantic Web And Beyond Computing for Human Experience, Springer, 2006, pp. 3–22.

[32] M. Szomszor, Dynamic discovery, creation and invocation of type adaptors for web service workflow harmonisation, Ph.D. thesis, University of Southampton (April 2007).

[33] M. Szomszor, T. R. Payne, L. Moreau, 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), Cardiff, UK, 2005.

[34] M. Szomszor, T. R. Payne, L. Moreau, Automated syntactic medation for web service integration, in: Proceedings of IEEE International Conference on Web Services (ICWS 2006), Chicago, USA, 2006.

[35] M. Szomszor, T. R. Payne, L. Moreau, Dynamic discovery of composable type adapters for practical web services workflow, in: Proceedings of UK e-Science All Hands Meeting, Nottingham, UK, 2006.

[36] C. Wroe, C. Goble, M. Greenwood, P. Lord, S. Miles, J. Papay, T. Payne, L. Moreau, Automating experiments using semantic data on a bioinformatics grid, IEEE Intelligent Systems 19 (1) (2004) 48–55.