



Department of Computer Science
The University of Auckland

Semi-automatic discovery of mapping rules to match XML Schemas

Sebastian Boßung

supervised by
Prof John Grundy and Prof John Hosking

November 25, 2003

Abstract

The conversion between two different XML languages that represent the same data differently is a common problem that occurs in many systems. The usual approach to solving this problem is to manually define a mapping that will convert one XML language into another. As this is both tedious and error-prone, it seems worthwhile to investigate possibilities of how to automatically discover the rules for these mappings.

This project presents a prototype mapper that attempts to map XML elements by looking at the names of the elements, their types and the overall structure of the XML Schema. It implements multiple algorithms to find matches for elements that are later combined to determine the final output. Because schema mapping can require a great deal of domain knowledge, an extensible architecture is presented that will accept plugins providing additional matching algorithms. These plugins are simple to develop.

Only in trivial cases will the resulting mapping be complete, some manual editing by the user is likely to be required. For this purpose a simple web interface is presented, but the prototype is constructed in a way that makes it easy to use as a component in other visual mapping tools.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Example	5
1.3	Related Work	6
2	Requirements	10
2.1	Actors & Roles	10
2.2	Use Cases	10
2.2.1	Create Mappings Semi-automatically	11
2.2.2	Import Schemas	13
2.2.3	Generate match candidates	13
2.2.4	Refine Mapping	14
2.2.5	Generate Output	14
2.2.6	Aquire Matching Suggestions	14
2.2.7	Browse Schemas	15
2.2.8	Install Plugins	15
2.2.9	Change Options	16
2.3	Functional Requirements	16
2.3.1	Match elements	16
2.3.2	Suggest matches for manual editing	17
2.3.3	Save matches for manual editing	17
2.3.4	Plugins	17
2.3.5	User Interface	17
2.3.6	Estimate quality of mapping	18
3	Design	19
3.1	Different Types of Mappers	19
3.2	Overall Architecture	19
3.3	Schema Representation	21
3.4	Output Format	22
3.5	Matchers	22
3.6	Match Datastructures	24
3.7	Refinement	24
3.8	Plugins	24
4	Implementation	26
4.1	Technologies	26
4.2	Schema Respresentation	27
4.3	Matchers	27
4.3.1	Partial Name Matcher	29

4.3.2	Levenshtein Name Matcher	29
4.3.3	Same Type Matcher	30
4.3.4	Synonym Matcher	30
4.3.5	Type Signature Matcher	31
4.4	Refinement	33
4.4.1	Refiner Architecture	33
4.4.2	Impossible Correspondences	34
4.4.3	Auto Resolver	35
4.4.4	Children Refiner	35
4.5	Plugins	35
4.6	User Interface	36
4.7	Output Format	38
4.7.1	Why not XSLT?	38
4.7.2	Own Schema	39
4.8	Options	39
5	Evaluation	41
5.1	Testing	41
5.1.1	General Note	41
5.1.2	Statistics	41
5.2	Simple Cases	42
5.2.1	Person Lists	42
5.2.2	Invoice Schemas	42
5.3	Different Bibtex XML formats	44
5.4	Auction Search Results	46
5.5	Usabililty of the Frontend	47
5.6	Complexity	48
5.7	Summary	49
6	Conclusions & Future Work	50
6.1	Conclusions	50
6.1.1	Architecture	50
6.1.2	Techologies	51
6.1.3	Algorithms	51
6.1.4	Summary	51
6.2	Future Work	52
6.2.1	Enhancements to the Framework	52
6.2.2	Enhancements to the Frontend	52
A	Acknowledgements	55
B	Schema for XML output	56
C	UML Class-diagram of the Framework	59
D	How to write extensions	61
D.1	Framework Architecture	61
D.2	Step by Step	61
D.2.1	The Plugin	61
D.2.2	The Matchers	62
D.2.3	The Schemas	62
D.2.4	Voting	63
D.2.5	Rules	63

D.2.6 Casting Votes	64
D.2.7 The Refiners	64
D.2.8 Options	65
D.3 Tips for Development	66
E How to use AXSM as a component	68
E.1 Important Classes	68
E.2 Step By Step	69
E.2.1 Running the Mapper	69
E.2.2 Getting the Results	69
E.2.3 User Interaction	69
E.2.4 XML Export	70
F The CD	71

List of Figures

1.1	Trivial example schemas	6
1.2	A VML mapping	7
1.3	Workflow of the Form Based Mapper	7
1.4	Form Based Mapper screenshot	8
1.5	Mapforce screenshot	9
2.1	System use cases	11
2.2	Screenshots of static prototype	12
2.3	Workflow when creating mappings	13
2.4	Screenshots of static prototype: Browse schema	15
2.5	Screenshots of static prototype: Options	16
3.1	Core Classes	20
3.2	The voting mechanism	20
3.3	UML diagram of the simplified schema representation	21
3.4	The correspondence datastructure	22
3.5	Output schema	23
3.6	Class diagram of the plugin infrastructure	25
4.1	Schema comparison	28
4.2	Example of simplified schema representation	28
4.3	Type Signature Matcher's datastructure	32
4.4	The Ballot class	33
4.5	Sequence diagram of refiner invocation	34
4.6	Class diagram of the plugin infrastructure	36
4.7	Trivial example: Screenshots	37
4.8	Output schema	40
5.1	Simple person list schemas	43
5.2	Statistics of the simple person list case	44
5.3	Invoice schemas	45
5.4	Statistics of the invoice case	46
5.5	Statistics of the Bibtextml case	46
5.6	Statistics on the auction search schemas	47
B.1	Output schema	57
C.1	Framework class diagram	60
D.1	Simplified schema representation	63
D.2	Correspondence datastructure	64

Chapter 1

Introduction

1.1 Motivation

Data is stored on different systems in a variety of different formats, which are assumed to be represented by XML Schemas here. Often these formats are used to hold corresponding information, which might need to be exchanged when the systems have to cooperate. This calls for a conversion between the formats of the two systems.

The traditional approach to this is to write program code on one or both systems that is able to convert the different representations of data into each other. Being a rather low level task that is highly related to the application domain, it requires programmers and domain specialists to jointly work on it. Obviously this work is tedious and error-prone and therefore a good candidate for which to investigate automation possibilities.

One step to simplify the task of building a mapping between schemas has been taken in the Formbased Mapper as described in [3] and [4] (also see section 1.3) by representing the schemas on both sides as business forms and allowing domain experts (i.e. business analysts) to make connections between them visually. Taking a closer look at the Formbased Mapper, one notices that some of the mappings are rather obvious to a human (i.e. because the elements have the same name). This leads to the idea of assisting the user by automatically discovering a part of the mapping. However, it is also not very difficult to find out that an automatic system will not be able to find mappings between all elements. This is because there is just too little information on the actual meaning of elements in the schema. Humans are able to find mapping rules by applying domain knowledge as well as the broad range of general knowledge they have acquired over the years. With complicated schemas even humans are unable to define a mapping without consulting a domain expert.

On some elements the program might be unable to find a mapping altogether, on others it might just not be able to decide between a few different options. Offering these candidate matches to the user can significantly help him with schema matching. Humans will also need to review the program's matches, but care should be taken to have as few false positives as possible.

1.2 Example

Figure 1.1 gives two schemas that are to serve as examples throughout this report. The schemas are trivial and thus small enough to print as well as understand at a glance. The lines denote correspondences between the respective elements on source

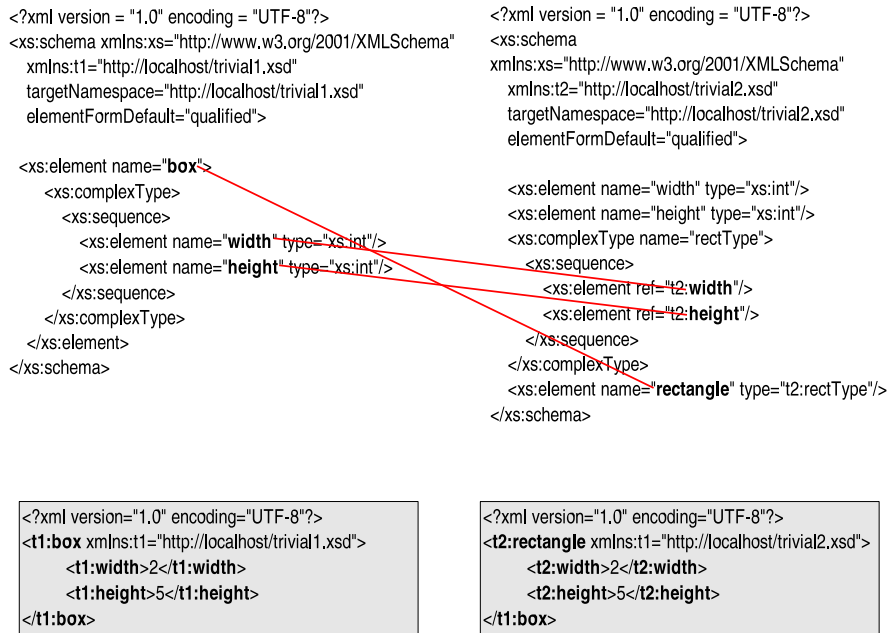


Figure 1.1: Trivial example schemas and an instance document for each.

(left) and target (right) side.

1.3 Related Work

As mapping data between different representations is a common task, quite a lot of research has been done on the subject. There are also many ways to approach the problem mainly differing in the targeted users (ranging from expert-programmers to complete non-programmers) and the degree of automation desired.

"Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems" [4] describes three visual languages to specify data mappings targeted at different audiences.

The first, VML, is geared towards the expert-programmer. It provides means to visually create mappings between two schemas. Contrary to most other approaches, there is no notion of a source or target schema. VML also dispenses with many of the differences between attributes and entities allowing easier mapping between them. The user specifies mappings by placing an inter-class definition between two entities. Attributes from the two entities are then mapped by equality, equation, function or procedural code.

The second, RVM, is made for people familiar with databases, who have experience in data structuring and formatting issues but none or very little in programming. RVM was developed to handle datastructures from the health sector. It introduces visual metaphors for mapping the hierarchical record structure. These include simple equivalences as well as conditional mappings using if-then-else structures. More complex mappings such as mappings between collections are also possible.

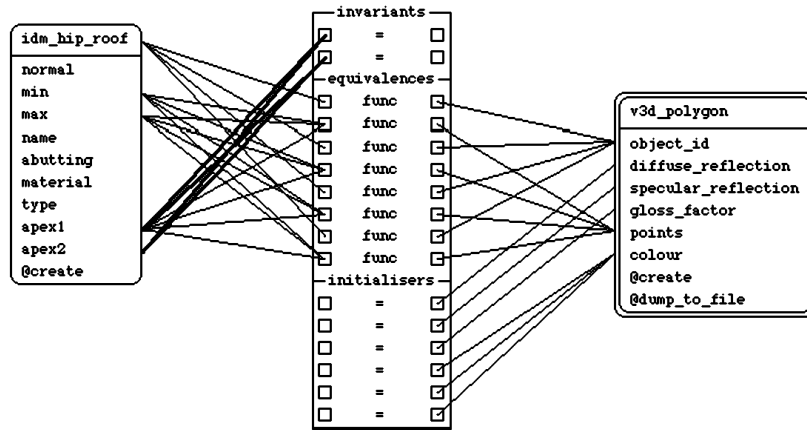


Figure 1.2: A VML mapping from [4]

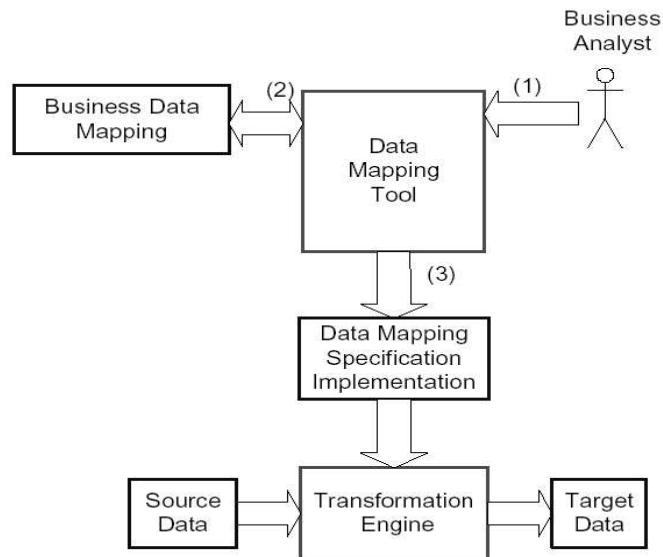


Figure 1.3: Workflow of the Form Based Mapper from [3]

Figure 1.4: Form Based Mapper screenshot from [3]

The third is the Form Based Mapper (FBM for short). It uses the notion of business forms to help non-programmers in specifying the mapping. The source and target schema are displayed next to each other and the user can identify matching element simply by dragging fields.

The FBM is the subject of Li Yongqiang’s master’s thesis [3] and described there in great detail. One possibility of specify input datastructures to the FBM are XML Schemas. After parsing the schemas, the FBM will present two business form like columns to the user. The user can specify mappings by dragging elements between these forms. The mappings are represented visually by connections between the respective elements. There is also support for more complex mappings (ranging from concatenation of two fields to the generation of nested structures) but this is handled in a less visual manner. To create an interface that is more familiar to the user, the elements of both forms are rearrangeable to resemble to paper forms more closely.

”A survey of approaches to automatic schema mapping” [1] gives an overview of various approaches to schema mapping in general. It deals with the overall architecture as well as details of the matching algorithms. The notion of hybrid and composite mapper architectures is taken from this article and discussed further in chapter 3.1. The article also deals with different input datastructures as it considers schema-level, element-level and instance-level approaches to schema mapping. Instance-level approaches look at the actual data in instance documents and compare instance documents conforming to one schema to those conforming to a second schema. Element-level approaches do not use the instance documents but the schemas. Matches are performed on single schema pieces (i.e. elements or attributes) only. Finally, the schema-level approach looks at larger structures in the whole schema (i.e. lists) to

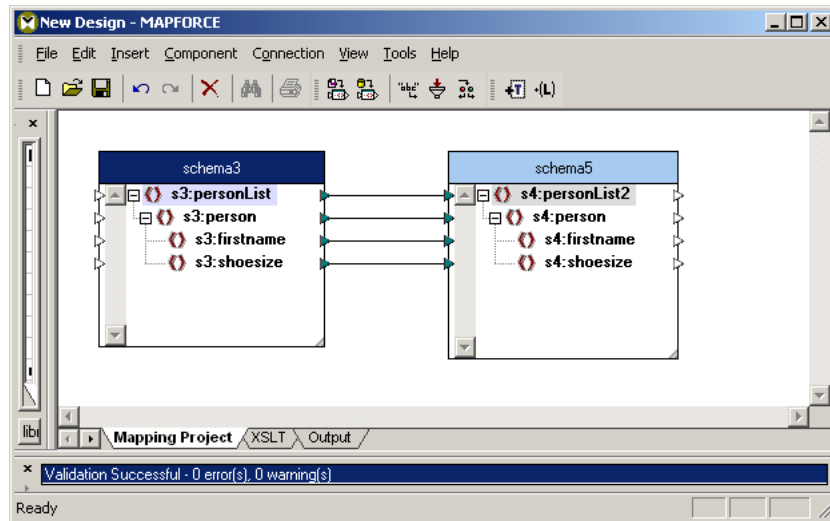


Figure 1.5: Mapforce screenshot showing mapping of two trivial schemas.

find matches.

The authors of "Automatic transformation of XML documents" [2] describe the Xtra system. Xtra attempts to automatically determine a mapping between two DTDs. This seems to address the very same problem dealt with here. However, basing Xtra on DTDs (because it was the industry standard at that time) limits the amount of information the system has available for matching. This means that it must mainly work on element names and their nesting. Having less information at hand also means that source and target schema have to be more similar. This is especially the case if one tries to find a complete mapping, as Xtra does.

There are also commercial visual mapping tools. One example is Mapforce by Altova [10]. It lets the user draw connections between elements of source and target schema. The tool will then generate XSLT that implements the mapping. There is also extremely limited support for automatic discovery of matches. This is limited by two conditions: the names of the elements have to be exactly the same (only case can be ignored) and they have to be direct sub elements to elements that are known to match. The types of the elements are also not considered (neither in the automatic match nor in the final XSLT). Mapforce will generate XSLT that attempts to copy the value of a string field on the source side in a target field that is declared to be an integer. This results in invalid instance documents. Handling type compatibility and conversion is thus entirely left to the user. Figure 1.5 shows a simple mapping in Mapforce.

Chapter 2

Requirements

The overall goal is to implement a prototype application that can automatically identify as many rules as possible that are needed to convert data between two XML schemas. This implies that requirement 2.3.1 is of primary concern. This prototype will be referred to as AXSM (for Automatic XML Schema Mapper).

2.1 Actors & Roles

The actors working on the system are:

- *Human user*: works directly with the system and chooses from the suggested mappings. Can also configure the system and install new plugins.
- *Other mapping system*: uses AXSM as a component to provide additional functionality to the end user. This could involve some kind of improved frontend or additional means to specify a mapping manually.

They have the following roles:

- *Manual mapper*: doing the basic steps to create a mapping. These steps are described by the use case "Create mappings semi-automatically". In addition to this, the manual mapper might wish to change the some matcher parameters to adapt the system to the specific task at hand.
- *Administrator*: Configures and administers the system. This mainly involves installing the application by preparing the correct environment and later putting in additional plugins.
- *Mapping system*: Uses AXSM as a component inside a larger system.

2.2 Use Cases

An overview of the use cases is given as an UML diagram in figure 2.1.

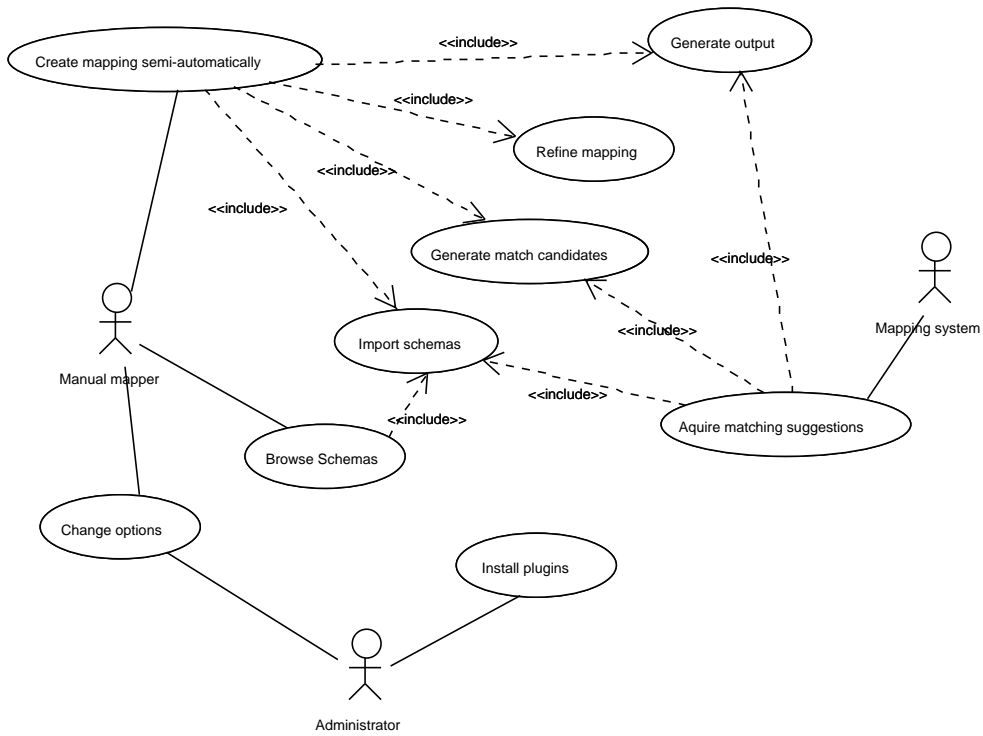


Figure 2.1: System use cases.

2.2.1 Create Mappings Semi-automatically

Use Case	Create mappings semi-automatically
Roles	Manual mapper
Priority	High
Description	Assist the manual mapper by giving suggestions on possible match candidates for the elements.
Includes	Import schemas Generate Match Candidates Refine mapping Generate output

This is the most common use case for the end user. It represents the normal workflow of specifying the input schemas, having the system generate match candidates, choosing from the candidates and generating the output.

Screenshots of an initial static prototype can be seen in figure 2.2. A diagram of the workflow of this use case is shown in figure 2.3.

Please specify a **source** schema. You can do so by either giving a URL pointing to one or by just pasting your schema in the text-area below.

URL:

The system has found some possible match candidates. Please indicate which ones are correct by checking the boxes on the right.

Namespaces:
 Source schema: s1: http://www.example.invalid/schema.xsd
 Target schema: s2: http://www.somewhere.invalid/import.xsd

Match	Votes	Correct?
s1:person -> s2:customer	2.0 - Total 1.0 - ExactTypeMatcher 1.0 - TypeSignatureMatcher	<input type="checkbox"/>
s1:orderList -> s2:orderList	3.0 - Total 1.0 - ExactNameMatcher 1.0 - PartialNameMatcher 1.0 - LevenshteinNameMatcher	<input type="checkbox"/>
s1:person -> s2:personList	0.6 - Total 0.6 - PartialNameMatcher	<input type="checkbox"/>

Here is the refined mapping as an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<axsm:mapping xmlns:axsm="http://localhost/axsm_out_1_0.xsd">
  <axsm:correspondences/>
  <axsm:candidates>
    <axsm:correspondence>
      <axsm:rule>
        <axsm:name>axsm.match.rules.ConvertRule
        <axsm:lossy>>false
      </axsm:rule>
      <axsm:source>
    ...
  </axsm:correspondence>
</axsm:candidates>
</axsm:correspondences/>
</axsm:mapping>
```

The result is also available as a [plain XML file](#).

Page loaded.

Figure 2.2: Screenshots of a static initial prototype.

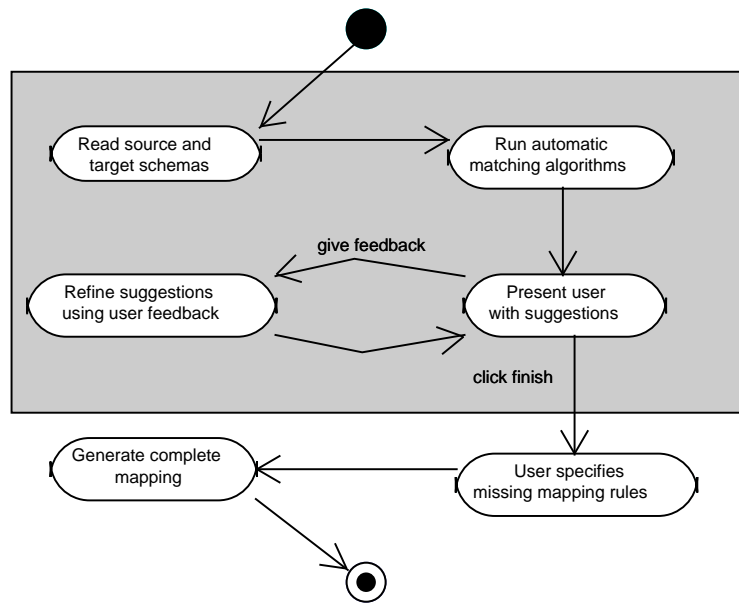


Figure 2.3: Workflow when creating mappings. Note that only the activities inside the grey box are covered by AXSM.

2.2.2 Import Schemas

Use Case	Import schemas
Roles	Manual mapper, Mapping system
Priority	High
Description	Inform the system about which schemas to use for source and target. Schemas can be given directly or by handing the system a URL that will be used to retrieve the actual schema
Includes	-

2.2.3 Generate match candidates

Use Case	Generate match candidates
Roles	Manual mapper, Mapping system
Priority	High
Description	This is the core functionality of the system. Given the source and target schema, run matching algorithms to find potential mappings for the elements. This is done autonomously without any user interaction.
Includes	-

2.2.4 Refine Mapping

Use Case	Refine mapping
Roles	Manual mapper
Priority	Medium
Description	As the user processes a suggested mapping, the system can draw conclusions on other mappings. The include eliminating now impossible mappings and reevaluating the likelihood of others.
Includes	-

Also see chapters 4.4 and 3.7.

2.2.5 Generate Output

Use Case	Generate output
Roles	Manual mapper, Mapping system
Priority	High
Description	Serialize the current state of the mapping into an XML file that can be easily imported by other applications.
Includes	-

2.2.6 Acquire Matching Suggestions

Use Case	Acquire matching suggestions
Roles	Mapping system
Priority	Medium
Description	In this use case AXSM is used as a component in another system that lets the user create a mapping. This other system relies on the automatic mapping capabilities of AXSM but does not use the frontend as the suggestions are presented to the user in a special user interface.
Includes	Import schemas Generate match candidates Create output

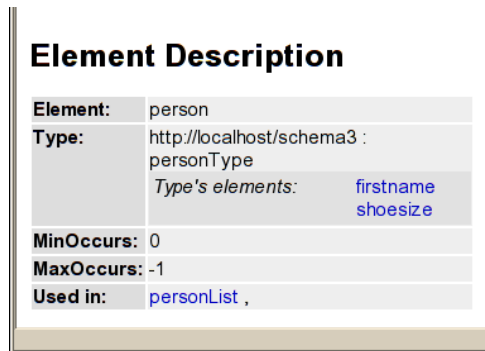


Figure 2.4: Screenshots of a static initial prototype: Browse schema.

2.2.7 Browse Schemas

Use Case	Browse schemas
Roles	Manual mapper
Priority	Low
Description	Let the user browse source or target schema from within the frontend. When an element is selected, its details are shown. These include element name, type, all parent and sub-elements. By clicking one of the elements the details page is focused on that element. This way, it is possible to move along the complete schema.
Includes	Import schemas

See figure 2.4 for a screenshot.

2.2.8 Install Plugins

Use Case	Install plugins
Roles	Administrator
Priority	Medium
Description	Plugins are used to provide new matching algorithms as well as additional domain knowledge. They come in one jar-file per plugin and can be installed by the administrator by putting this file in a specific directory.
Includes	-

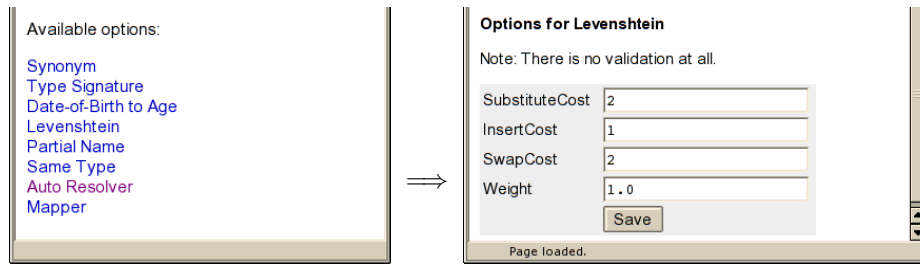


Figure 2.5: Screenshots of a static initial prototype: Options

2.2.9 Change Options

Use Case	Change options
Roles	Manual mapper, Administrator
Priority	Low
Description	Many matching algorithms will have parameters. Changing these can allow the user to adapt the system to a specific situation and achieve better results. As the core framework does not know about all possible parameters of future plugins, parameters have to be discovered automatically.
Includes	-

Changing parameters happens in two steps: First the user chooses the object the parameters of which should be edited. In the next screen a list of these parameters is presented. They can be changed and saved. Figure 2.5 shows both screens. After changing parameters, the user will have to rerun the mapper for the results to take effect. This could be facilitated by providing a direct link to rerunning on each page.

2.3 Functional Requirements

2.3.1 Match elements

While XML schema contains both types and elements, the match between two schemas has little practical use in itself. Rather it will later be used to convert a source XML instance document that adheres to the source schema into another instance document as specified by the target schema. This calls for a mapping to be defined between elements.

This prototype is to do this using two different approaches: Firstly, looking at the elements themselves, secondly also considering their types.

A major source of information for mapping elements are their names. Therefore different approaches down this road will be pursued. Obvious ones are exact name matching, name matching based on edit-distance (also see chapter 4 on edit-distance). However, one can also try to incorporate more outside knowledge into the matching process by looking for synonyms using a standard or domain specific dictionary of synonyms.

Types provide valuable information in XML schema. This is because there are (a) quite a few simple types which limits the possible candidates and (b) complex types are rich in structure allowing for multiple ways to exploit this in matching. One more thing is that source and target schema could actually make use of the same

types. This is certainly true for the built-in simple types but also for user specified complex ones (by referencing a common third schema). Match candidates can thus be discovered by exact type matching. Another approach is to look for types in the target schema that have a signature¹ that is as similar as possible to the source type.

2.3.2 Suggest matches for manual editing

Even if some elements cannot be mapped, the system might have some information about them. This information should not be discarded but given to the user to assist him in manual mapping. An example for this is the case where the system has multiple match candidates and cannot decide, which one is the best match. Instead of being presented with the whole range of possible elements from the target schema, the user would only have to choose from the provided candidates.

2.3.3 Save matches for manual editing

The system needs to be able to export the discovered matches for further processing with other tools. The focus of this system is on semi-automatic mapping and it is unlikely that a complete mapping can be discovered in this manner. Therefore the user will have to employ additional tools that allow manual specification of the missing mapping rules.

To make all suggestions useful in further processing, the output format should be able to export sets of suggestions per element and not just the most likely suggestion or matches that the user actually marked as correct.

The format used here has to be interoperable. Preferably it should be possible to handle it with existing tools and libraries. Therefore XML seems to be a good choice. To make sure the developers of other tools know what they are dealing with, the format should be described by an XML Schema.

2.3.4 Plugins

Mapping between schemas works best when the one is familiar with the domain the schemas come from and has a broad scope of general knowledge as well. This is just as true for machines as it is for human users. Manual mapping will generally involve a domain expert who will bring the necessary knowledge into the process. During automatic mapping there is no such expert. This means that all the knowledge in the system has to be put in at design time. Obviously it is impossible to do so for every conceivable domain.

To make the system more flexible and to allow it to adjust to various domains, it is necessary to have some mechanism of extending the functionality as needed. This should allow the user to write his own matchers providing both domain knowledge and additional general matching algorithms. The new matchers are bundled in plugins, which should be easily installable in the system.

2.3.5 User Interface

A simple, web-based user-interface is to be developed as well. While not the main focus of this project, it should guide the user through one complete workflow of using the prototype. First, it queries for source and target schemas. The user can either specify those directly by pasting them into a textarea or give a URL. Then the mapper

¹The signature of a type is comprised of types and cardinalities of its subelements. This will be defined in a more formal manner in chapter 4.

is set to work on these schemas and the resulting mapping candidates are presented to the user. The user can make choices which ones are correct but can also leave some decisions open. The list of candidates should of course be refined while the user chooses candidates to show only those that still make sense. Last, the frontend displays the resulting XML document and also makes it available for download as a file.

2.3.6 Estimate quality of mapping

One aspect of the quality of a mapping is how many wrong rules are in the collection of automatically detected ones. For each generated mapping rule the system should therefore be able to give a certainty of how likely it is that this rule is correct. Another important point is what proportion of the elements the system was actually able to match. These values should be reported to the user to give a rough idea of how much manual editing will be needed to make the mapping complete.

Chapter 3

Design

3.1 Different Types of Mappers

Rahm and Bernstein [1] distinguish between two types of mappers: hybrid and composite ones. A hybrid mapper integrates different approaches of matching elements into one algorithm, while a composite mapper has multiple separated algorithms whose results are combined after running each of them individually. In the hybrid mapper synergies that can exist between the algorithms can be exploited. This will allow the hybrid mapper to run faster. On the other hand the composite mapper keeps the algorithms separated such that they know nothing about each other. This has several benefits: the user is free to run any selection of algorithms in any order he chooses, the weighting can be changed rather easily¹, and it possible to extend the mapper with further algorithms without having to neighter understand nor touch the existing ones.

In this project a prototype of a mapper is to be implemented. Because the project is rather limited in time, extensibility is very important. Therefore the basic architecture of a composite mapper was chosen. As the prototype will not have all desirable algorithms implemented this seems to be the most sensible approach.

3.2 Overall Architecture

In the composite mapper architecture each algorithm that is used to find matching elements is implemented separately. Thus the central points of the architecture is the question of how the individual algorithms are implemented, how each is called and how the results are combined later. The central classes are shown in figure 3.1.

Here each algorithm is implemented a class derived from the abstract class `Matcher`². Each matcher will implement a method called `findCandidates` that takes the whole schema as a parameter and return a collection of `ElementCorrespondences`. The reason for passing the whole schema rather then each element individually is twofold: some algorithms might need global knowledge about the schema (such as the type matcher) and also might be able to treat the whole schema more efficiently than it would be possible by looking at each element on its own.

¹This is not the case with the hybrid mapper, as the output of one algorithms might implicitly serve as input to another.

²Here an individual algorithm is called `matcher`, while the whole prototype is a `mapper`. These two terms should not be confused.

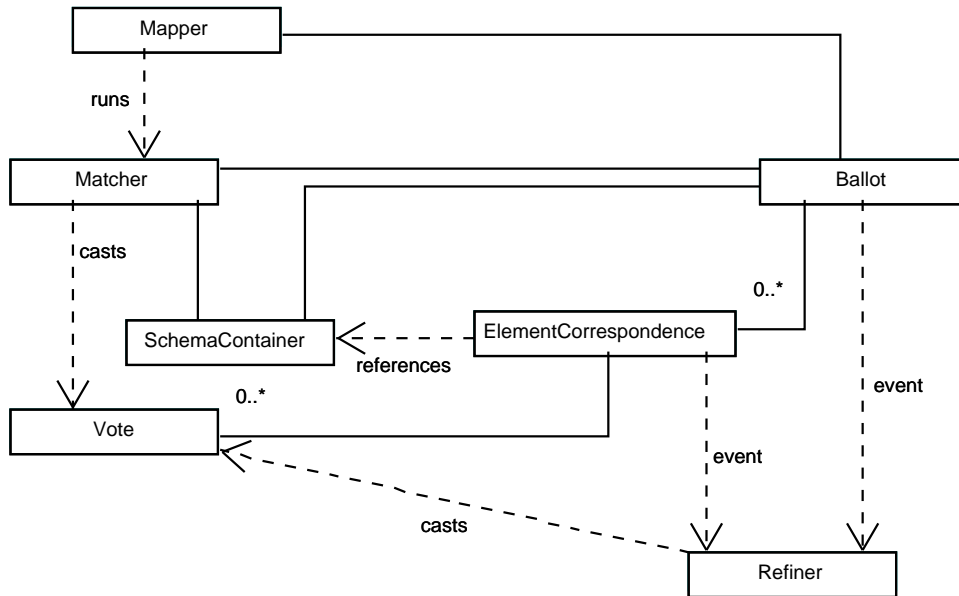


Figure 3.1: Core Classes

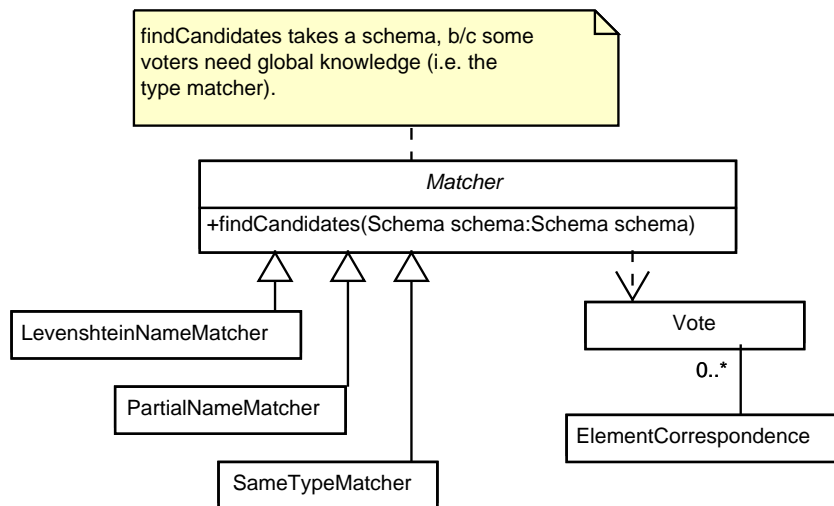


Figure 3.2: The voting mechanism

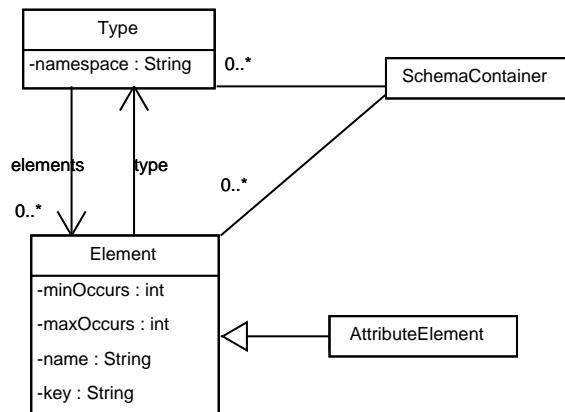


Figure 3.3: UML diagram of the simplified schema representation

Each matcher attaches a Vote to the ElementCorrespondences it is favouring. The Vote has a field certainty where the matcher can record how sure he is of his suggestion.

The output of the matchers is aggregated in a class Ballot and the Votes are later used to decide which ElementCorrespondence will be settled for. If the system cannot decide on one, all plausible ones are passed on as suggestions.

While the user works with the suggestions, the system uses the given input (i.e. the fact that the user declare a suggestion to be correct) to refine the set of suggestions. This is done by refiners that react to change events from the ballot or an individual correspondence. This will be discussed in more detail in section 3.7.

From the point of view of the user the system is both a framework and a component. He can use plugins (see sections 2.3.4 and 4.5) to extend it with further matching algorithms or domain knowledge. In this case the framework aspect is used. When designing tools to visually map XML Schemas, the system can be used as a component that takes two XML Schemas and after one method call returns an XML document of match candidates. These candidates can either be used directly by the bigger system or in conjunction with this system's refinement capabilities.

3.3 Schema Representation

The input format to be used for the mapper is XML schema (see the W3C's website [5] for details). The schema parsing parts of the Castor project (see [6]) are used to convert the schema into a Java object representation. This representation is very close to the schema standard. This means that it has some properties that are not helpful to the matching tasks. This is for example the notion of attributes. You can represent any attribute as an element that only occurs once. This is just a matter of notation. Also XML schema's choice and sequence particles can be nested in very complicated ways. It is not very sensible to assume the same kind of nesting on the other side of the transformation, as there are multiple ways to specify equivalent or very close cardinalities for the elements.

For these two reasons this prototype uses a simplified mapping as depicted in the UML diagram 3.3. Here no nesting of choice and sequence is considered but all

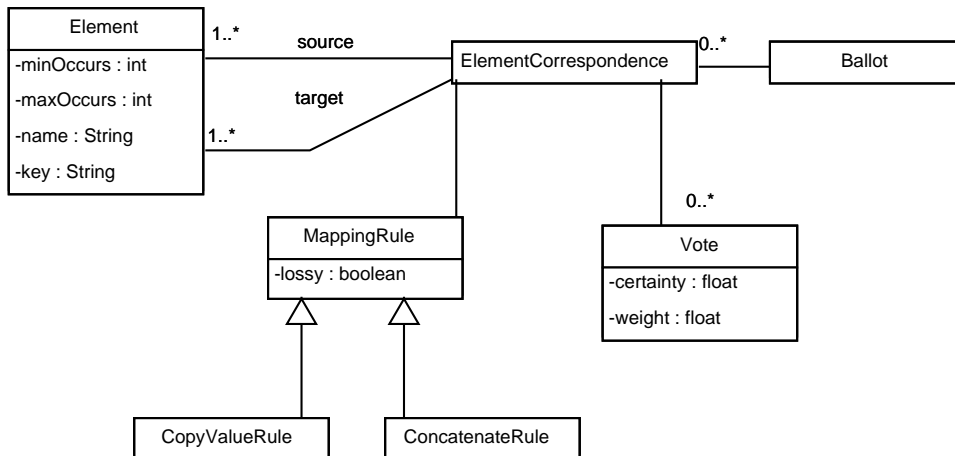


Figure 3.4: The correspondence datastructure.

elements are just brought up to the first level within their type. The cardinalities are adjusted to reflect their new status of an individual element. Note that this is not exactly equivalent but seems sensible for the reasons mentioned earlier. On top of this, attributes are represented as special kind of elements. This seems sensible as well because one cannot assume that the target schema will use element or attribute normal form (or anything inbetween).

3.4 Output Format

As automatic discovery will only yield a part of the required rules to map two schemas, using the output of the automatic mapper as input for other visual mapping tools is an important scenario.

There seem to be no standardized formats that are really suitable for storing high-level mapping information. While storing the discovered rules using XSLT certainly describes them well, it is almost impossible to parse XSLT when importing the mapping information into another tool. The definition of an appropriate XML language to store the mapping rules discovered as well as possible candidates the automatic mapper was unable to map will be necessary.

3.5 Matchers

In the composite mapper architecture each matcher encapsulates an algorithms that searches the schemas for potentially corresponding elements. This means that the matchers run independently and cannot rely on each other for clues about candidates.

The matchers implemented in this prototype fall in two categories: those working on element names and those working on the types of the elements. There is no matcher that works on a hierarchie of elements, because technically speaking there is no hierarchy of elements in an XML Schema. Rather, each element has a type that can again contain elements. A hierarchical matcher would therefore be a type matcher that looks at more than one type at a time (i.e. one detecting composite patterns).

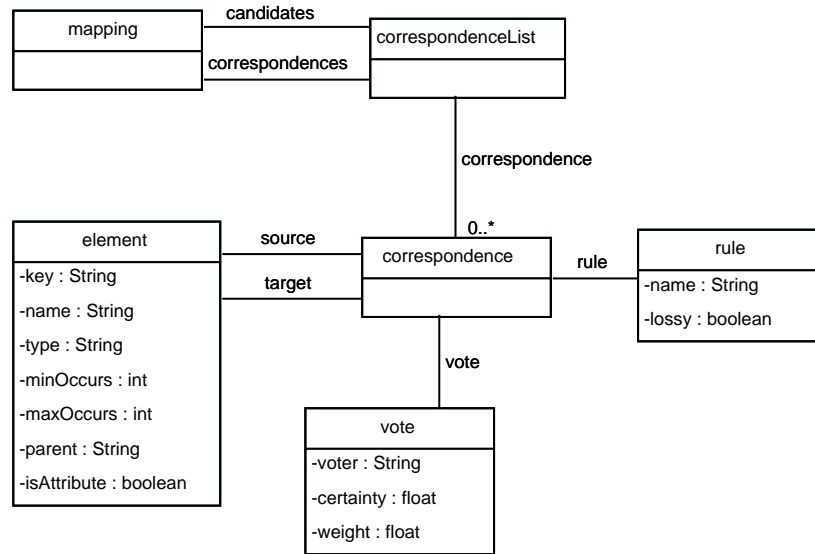


Figure 3.5: Output schema. Everything shown here as attribute is actually implemented as elements (with the exception of `isAttribute` in `element`). An element "mapping" of type `mapping` is the root element

There are three name based matchers. Type based matchers work on types and in the end vote for element correspondences between elements that are of these types.

- *Levenshtein name matcher*: Will compute the Levenshtein distance³ between every pair of element names from the source and target schema and vote for those pairs to be corresponding whose Levenshtein distance is below a certain threshold.
- *Partial name matcher*: Checks for every pair of element names whether one name is part of the other.
- *Synonym matcher*: Uses a dictionary of synonyms (such as WordNet [7]) to find element pairs whose names are synonymous.

And two type based ones:

- *Same type matcher*: Looks for elements of exactly the same type. In a lot of cases these will only be elements of the simple types built into XML schema. However, as the two schemas will probably be from the same application domain, they might both use application specific types that are defined in a third schema.
- *Type signature matcher*: Will look for element pairs that are not of the same type but whose types are similar. Similarity here is taken to mean that a type has a set of elements that (approximately) matches that of another.

The mapper gathers the matchers to be run from all plugins that can be found. After this, a loop is run, that executes each matcher. This happens in no specific

³The Levenshtein distance is also known as the edit distance. It is defined by the number of edit steps it takes to convert one string into another. Edit steps are be delete, insert or substitute.

order, so there is a real need for the matchers to be independent. Running a matcher is done in two steps: The matcher is initialized by passing it the datastructure to work on (the ballot is set). Then the actual algorithm is run.

Another type of matcher is the data based matcher. It looks at actual instance data and tries to find corresponding elements by analyzing it. This however requires a complex infrastructure that is able to hold multiple XML document instances simultaneously and allows to run functions (i.e. computing the average of numerical fields) on this infrastructure. Implementing this was not achievable in the time given. A few ideas are discussed in chapter 6.2.

3.6 Match Datastructures

All matchers work on a common datastructure, that collects their results and later makes it possible to derive suggestions for the overall mapping. This structure is depicted in figure 3.4. The central class is `ElementCorrespondence`. Each instance describes a correspondence between source and target elements. Note that these do not have to be single elements, but can be lists on the source as well as on the target side. The Rule that is attached to each `ElementCorrespondence` describes the kind of relationship. Matchers vote for an `ElementCorrespondence` by adding their vote to the list. This vote carries a certainty factor, reflecting how sure the matcher is of its suggestion. There is also a weight factor giving the weight the user assigned the matcher. All `ElementCorrespondences` are collected in a `Ballot`. This `Ballot` will later also provide methods needed to handle the `ElementCorrespondences` in common use cases efficiently.

3.7 Refinement

Once the mapper has run all the matchers, the system will present the user with a list of suggestions of how to map each element. As the user makes the first decisions on which of those suggestions are correct, valuable information is given to the system. Marking certain correspondences as correct makes others impossible and still others more likely.

There are mainly two cases:

- *Impossible cases*: If an element from the source is already mapped onto a certain element in the target schema, it cannot be mapped at the same time onto another element from the target schema. Considering that this also works in the opposite direction allows the system to drop some suggestions every time the make a choice. This is a simplifying assumption and will be discussed further in chapter 4.4.2.
- *Increased likelihood*: When the user marks a mapping between two elements as correct, mappings between the child elements become more likely than they were before. This might result in a reordering of suggestions with the now more likely ones further to the top.

Matchers are grouped in plugins, this is depicted in figure 3.6.

3.8 Plugins

Two things are important for a plugin interface to AXSM: The interface has to be simple and the user has to be able to install plugins easily. The first can be achieved

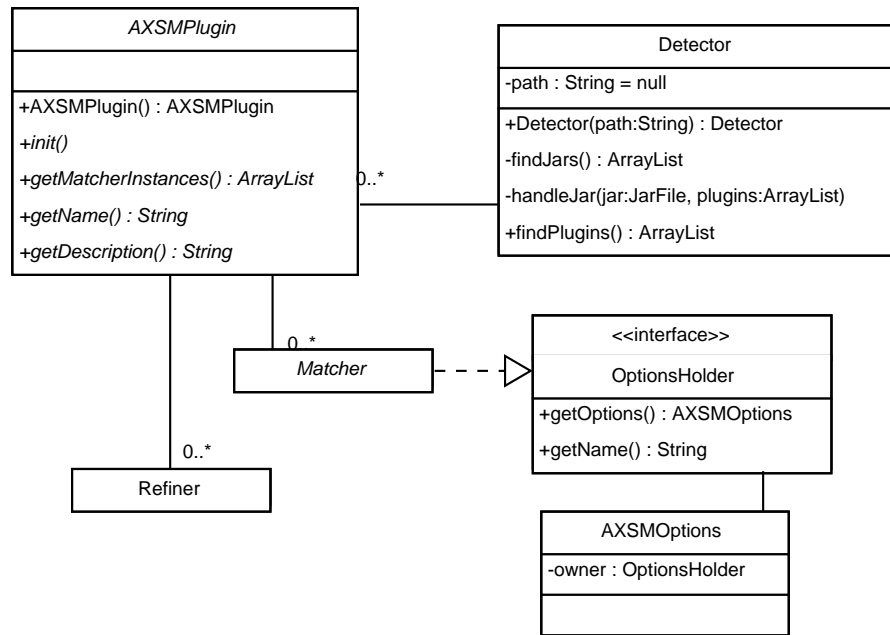


Figure 3.6: Class diagram of the plugin infrastructure

by having a limited set of interface classes, that are well documented and do not contain more than what is needed to write plugins. This helps plugin developers because it is not necessary to understand the complete system, just the plugin classes will suffice. Secondly, deploying a plugin to be used by the system should not involve complicated steps. It certainly must not involve any changes to the code. Ideally, plugins can be changed at runtime.

The first can be achieved by proper design of the classes. The setup in this system is depicted in figure 3.6. There are of course some more classes needed to represent the mapping results, these are shown in figure 3.4 and have been discussed previously.

Dynamically loading code is not very difficult in the Java platform thanks to the reflection mechanism and a variety of different classloaders. It is thus possible to discover plugins that are packaged in jar-files at runtime. This makes deployment for the user very simple, all that is necessary is dropping the jar-file in a specific directory.

Having visualization plugins that allow the user to chose between different ways of displaying the system's suggestions is also very interesting. However, this is beyond the scope of this project and therefore discussed briefly in the future work section 6.2.

Chapter 4

Implementation

4.1 Technologies

This section briefly discusses the technologies chosen for various parts of the system. Often these choice were heavily influenced by the personal tastes of the author and his familiarity with certain technologies (or rather his unfamiliarity with others). This is true for the implementation language and the frontend, but not for input and output format.

Java was chosen as the implementation language for the framework for a couple of reasons. To make sure that it can later be used by as many people as possible, it is necessary to implement in a well known language, which Java certainly is. Java also has two capabilities that were of particular importance in this project: object-orientation and reflection.

Supplying a plugin infrastructure as well as a simple component interface is much easier in a object-oriented language than i.e. in a procedural. Inheritance and late binding make it possible to hide acutal implementation from the user of the component interface. They also allow development of generic algorithms to handle plugins.

The other importatant feature of Java is reflection. Using this facility it is possible to introspect code at runtime. This can be used to handle classes that were not known at implementation time or to create algorithms that only make minimal assumptions about the used classes. In this system reflection is important for handling the option JavaBeans (see section 4.8) and discovering the plugins (see section 4.5). In the options case reflection is used to discover the fields of the JavaBean, each of which represents a changable option to be presented to the user. Plugins are found and loaded using the URLClassLoader, that can load classes from jar-files.

The frontend presented here is implemented as a web application with the help of the Struts framework. Tomcat is used as a web container. Together these make it possible to develop a simple frontend with very little effort. Struts and Tomcat are both freely available as Apache projects making it easy for others to run the frontend.

XML technologies are used for input to as well as output of the system. XML Schema gives the input. This gives a description of the datastructures that can be very specific, which is important because one needs as much information as possible to match elements. The predecessor technology of XML Schema, DTD, mainly provides structure and gives next to no information on datatypes. However, types are very important as they are the only way to match elements if one cannot find a match solely based on element names. They also provide extra confidence if a name match is possible.

A popular format to specify translation of XML documents is XSLT. It is a powerful language to create these mappings. However, for use in this system it is not suited. The output of this system needs to be parsable by other system, which is difficult in XSLT. Also see section 4.7 on this issue.

4.2 Schema Representation

XML Schema is a powerful yet flexible way to define an XML language. In a schema a great number of aspects of the language can be defined. On the one hand, this is helpful to the matching algorithms because it gives them a lot more information than i.e. a DTD does, on the other hand, in some parts a schema can be overly specific or there can be different ways to express the same concept. These cases are hindering the matching. Thus, a special representation of the schema was devised that is very close to the original XML Schema. Figure 4.1 depicts XML schema and the simplified representation used here. The differences are:

- *Single type representation:* XML Schema makes a difference between simple and complex types. This is not necessary in the matching process and thus a representation with just one kind of type is used to simplify the matchers.
- *Flattening particles:* In XML Schema one can use choice and sequence groups (also called particles or model groups) to specify the elements of a complex type. These particles can be nested allowing very flexible ways of giving ordering and cardinalities of the elements. However, this also means that there are many ways to specify the same or very close cardinalities of elements. In addition to this, the ordering of elements is usually not significant from a content point of view. For these reasons it does not seem to make sense to assume that the target schema will use the exact same ordering and nesting as the source schema. All elements in a complex type are therefore pulled up to the first level (all nesting is flattened) and given cardinalities that correspond to the effective ones in the original schema.¹
- *Normalize attributes to elements:* XML Schema can have attributes defined in complex types that are attached to elements in the instance documents. Without loss of any data, every attribute can be transformed into a (sub-)element. The cardinalities of that element are either $[0 \dots 1]$ (if the attribute is optional) or $[1 \dots 1]$ (if it is required).

Assuming that the target schema will use attributes or elements in the same places as the source schema does not make much sense because attributes and elements are interchangeable as described above. Therefore attributes are represented as elements in the simplified schema, albeit of a special kind.

4.3 Matchers

Matchers roughly fall in two categories: Name and Type. The first category works on the element names. This usually involves comparing names from source and target schema with each other and deciding on likely matches. The second category looks at the types of elements. This can either mean working on the declared type (given by

¹Doing so can be a problem in some cases if ordering of elements is used to implicitly define associations between them. As this does not seem to be a particularly good design nor occur too often, this case is ignored.

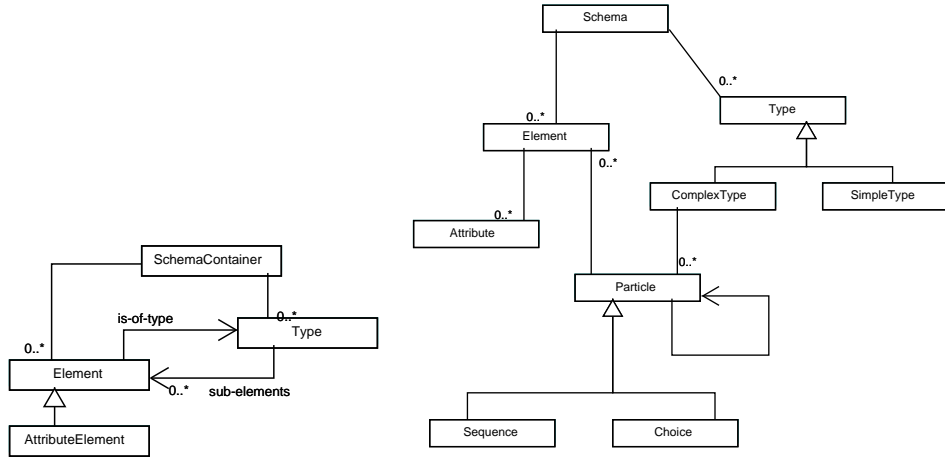


Figure 4.1: Schema comparison: Simplified schema representation (left), XML Schema representation (right, in reality it is even slightly more complicated)

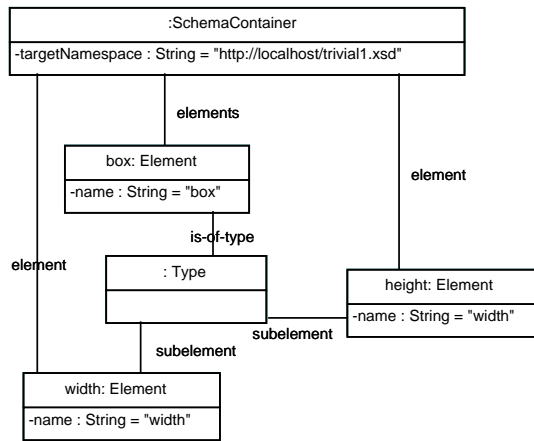


Figure 4.2: An example of the simplified schema representation using the first schema from chapter 1.2.

namespace and type name directly) or looking into that type and drawing conclusions from its structure.

The abstract super-class of all matchers (the class `Matcher`) has a method `normalizeName(String)` that will convert the passed name to lower case if the property `caseSensitive` is not set on the matcher. It could be extended to also do other things such as dropping hyphens. Parting names into individual words might also be useful. I.e. when the source schema uses camel-case "customerBillingAddress" and the target schema some other variant "customer-billing-address", both could be normalized into "customer billing address".

4.3.1 Partial Name Matcher

The Partial Name Matcher takes each element from the source schema and looks for elements in the target schema who's names contain source element's name. This is also done the other way around. The idea is to find correspondences in cases where the element names in one schema have been pre- or postfixed. The Partial Name Matcher will of course also find elements that correspond exactly by name, so there is no need for an Exact Name Matcher.

The vote of the Partial Name Matcher depends on the overlap of the two names. Obviously, a longer overlap should result in a higher vote. On top of that, the overall length of the names should also be taken into consideration, because a five character overlap is a lot on a six character name, but very little on a thirty character one. To achieve this, the certainty $c \in [0 \dots 1]$ in the votes is calculated as:

$$c = \begin{cases} s/t & \text{if } s < t \\ t/s & \text{else} \end{cases} \quad (4.1)$$

Where s is the length of the source name and t the length of the target name.

4.3.2 Levenshtein Name Matcher

Sometimes element names that should be exact name matches are not because the names differ slightly. This can be due to mis-spellings or the use of different legal spelling variants such as in "colour" and "color". To match elements by name under these conditions a notion of similarity between strings is needed. This comes as the Levenshtein distance (also known as the edit distance). Every insertion, deletion or substitution is charged a certain cost (usually 1 for all kinds of edit steps). The cost of transformation with the smallest total cost is called the Levenshtein distance.

To make use of this, the Levenshtein distance is computed for every possible pair of element names. If it is below a configurable threshold, the matcher will vote for the combination. The certainty of the vote needs to take into account how different the names are. However, as with the Partial Name Matcher, it should not be forgotten that lengths of names can vary greatly. Also accounting for this, the certainty c of two names with length s and t having a Levenshtein distance of d is given by:

$$c = 1 - \frac{d}{\min(s, t)} \quad (4.2)$$

Note that this allows negative votes if the distance is small but the lengths of the two names vary greatly and some edit steps are assigned a cost > 1 . Thus one has to take care when adjusting the cost parameters.

4.3.3 Same Type Matcher

This matcher looks at all possible combinations of source and target elements and votes for them to correspond if they are of the same type. This is fairly easy to establish, as name and namespace URI identify a type uniquely in XML Schema. Comparing these two is thus enough to establish a correspondence. For this type of correspondence the Same Type Matcher attaches a certainty of 1.0 to the vote cast.

In addition this, it will also vote for elements to correspond if the type of the source element can be transformed losslessly into the one of the target element. This is the case when the source type is a number and the target type string. Boolean as well as date and time types also seem to be convertible losslessly, but you have to make some assumptions about the format (what notation is used for "true" and "false", what date representation respectively). In this case the certainty of the vote will be 0.3.

To sum up, the certainty of the Same Type Matcher's votes is:

$$c = \begin{cases} 1 & \text{if elements are of identical type} \\ 0.3 & \text{if elements are of losslessly convertible type} \end{cases} \quad (4.3)$$

4.3.4 Synonym Matcher

When two schemas are developed independently, different people are likely to use different but synonymous words for the same thing. This might yield schemas with very similar structure but seemingly (to the Partial Name Match for instance) unrelated element names. However, somebody understanding the language used, will be able to identify many of the element names as synonymous. To automate this process, the WordNet lexical reference system is used that organizes English words into synonym sets [7]. This does have some serious limitations compared to a human doing the synonym matching: It only contains English words and does not have synonyms from specific application domains. However, it allows a wide variety of general English synonyms to be identified.

The WordNet system is compiled into a platform specific set of applications but available as C source so that it can be compiled for a lot of target platforms as needed. There is also a pure Java interface library Java WordNet Library (JWNL) [9] that is used in this implementation. It relies directly on the WordNet dictionary files and avoids the need of any native code.

The Synonym Matcher iterates over all the element names from the source schema. For each element name the following processing steps are performed:

1. The linguistic base form of the word is found using functionality from WordNet.
2. All synonyms are obtained from WordNet.
3. The target schema is searched for elements that have a name from the synonym list.
4. If such an element is found, the Synonym Matcher will vote for a correspondence between this element and the current source element.

In WordNet the returned synonyms are grouped by senses of the original word. This does not help the Synonym Matcher as it does not understand the sense. The grouped list is therefore transformed into a flat list and all duplicates are deleted.

The Synonym Matcher is implemented in a plugin to demonstrate the use of the plugin interface in AXSM (see 4.5 for details on plugins).

4.3.5 Type Signature Matcher

The signature of a type is a vector holding (type, minOccurs, maxOccurs) of each sub-element of the type. The vector is unordered, as the ordering of the elements in a type is assumed not to matter (see section 4.2)². The Type Signature Matcher is different from the Same Type Matcher because the latter looks at the type of the elements under consideration while the former looks at the sub-elements of their types (aka. the signature).

To find out how similar two type-signatures are, an algorithm is run that tries to find matches for as many subelements as possible looking at type and cardinality of their elements. Here is the detailed algorithm:

```
function similarityRecursive(sourceElements, targetElements) returns similarity
begin
  if size of sourceElements = 0 then return 0
  let value be the best similarity-value
  remove first entry from sourceElements and assign it to sourceElement
  for all targetElement in targetElements do
    newValue := similarityRecursive(sourceElements, targetElements\targetElement) +
      elementSimilarity(sourceElement, targetElement)
    if newValue > value then value := newValue
  rof
  return value
end
```

```
function elementSimilarity(sourceElement, targetElement) returns similarity
begin
  if correspondsTo(sourceElement.type, targetElement.type) then result += 0.7
  if sourceElement.minOccurs >= targetElement.minOccurs then result += 0.15
  if sourceElement.maxOccurs <= targetElement.maxOccurs then result += 0.15
  return result
end
```

```
function computeSimilarity(sourceType, targetType) returns similarity
begin
  return similarityRecursive(sourceType.elements, targetType.elements)
end
```

This algorithm is very similar to computing a perfect match of bipartite graphs. The notion of edges in the graph would not be binary (there is an edge or there is no edge between two vertices) but the graph would be complete with the edges being assigned values (as returned by *elementSimilarity*). The perfect match is then the match with the highest total value of the edges used. However, the implemented algorithm also tries whether not matching an element might result in a better total similarity. This is not shown in the pseudo-code to keep it more understandable.

The actual implementation also includes some shuffling of lists to avoid re-instantiation or cloning (see the CD for code).

The overall approach taken by the Type Signature Matcher is an iterative one. Iteration is done, because there are dependencies between types (type A might contain

²This refers to the sub-elements of a *type* in the schema, not the element ordering in an *instance document*, that might well make a difference.

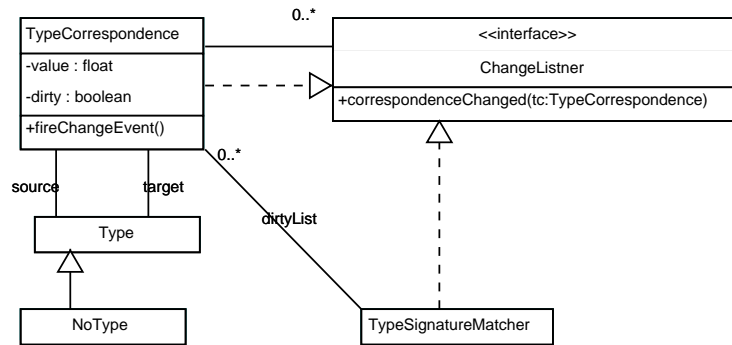


Figure 4.3: Type Signature Matcher’s datastructure. Both TypeSignatureMatcher and TypeCorrespondence implement ChangeListener. The first needs the events to add TypeCorrespondence to the dirty list. The second uses the events to listen to the TypeCorrespondences it is depended upon.

an element of type B, so B is dependent on A), in the worst case even circular ones. To account for this, the matcher does not walk the tree of types from bottom to top³ but just handles the list of types obtained from the schema linearly. In the beginning the corresponding types are only known for the primitive types built into XML Schema and possibly other ones that come from a third schema common to both source and target schema (these can be discovered by looking at the namespace URI and name of a type).

The matcher will set up a dependency graph of type correspondences . This allows marking a correspondence record as ”dirty” whenever its target type’s correspondence record has been updated (see figure 4.3 for an illustration of the datastructure used). The dirty correspondences are added to a dirty list. Before doing any matching the matcher will first add every type correspondence to that list. An iteration is then done by simply going over the list and trying to find a better match (hoping that the additional information aquired in the meantime will allow matches that were previously not known). Finish iterating when the correspondences cannot be improved over the last iteration (the total value of all type correspondences does not increase any more). This is automatically the case if the dirty list is empty.

The Type Signature Matcher will now vote for elements that are of corresponding type (according to its list of type correspondences). The certainty of the vote cast is:

$$c_{parent} = \begin{cases} \sigma & \text{if } \sigma > \tau \\ 0 & \text{else} \end{cases} \quad (4.4)$$

Where $\sigma = \text{computeSimilarity}(\text{sourceType}, \text{targetType})$ and τ is a configurable threshold. A vote with the certainty 0 is not cast.

The Type Signature Matcher calculates a similarity between two types. To do this it looks at the lists of sub-elements of the source and target elements and tries to find the best match between them (as explained above). The output of this algorithm is valuable beyond providing information whether the source and target elements are a match. The best match of the sub-element lists should also be used in the overall mapping. To achieve this, the Type Signature Matcher votes for correspondences

³Note that the ”tree” might contain loops and thus not be a tree at all, therefore the bottom-to-top approach will not work in the general case.

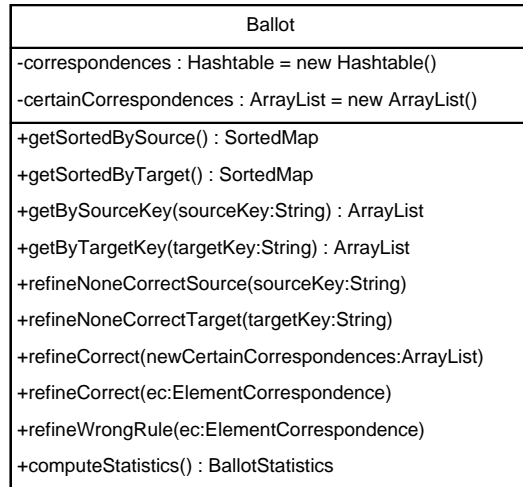


Figure 4.4: The Ballot class. This representation does not show all methods.

between sub-elements if the similarity of the parent elements is above the configurable threshold τ . The sub vote used is also adjustable by setting ν .

$$c_{children} = \begin{cases} \nu & \text{if } \sigma > \tau \\ 0 & \text{else} \end{cases} \quad (4.5)$$

4.4 Refinement

As discussed in chapter 3.7 reacting to user input can be useful in trying to speed up the interactive mapping process. To achieve this end, three extensions are implemented in the AXSM framework: A general algorithm that drops now impossible correspondences on each input, an Auto Resolver that automatically marks correspondences as correct if they meet certain criteria and a Children Refiner that accounts for the fact that matches between sub-elements of elements that are known to correspond are likely to correspond as well. Plugins can specify further refinement algorithms that react to user input and work on the system's suggested matches.

4.4.1 Refiner Architecture

There are two hook points for a refiner in AXSM. It can be notified every time the user decides that a suggested correspondence is correct by implementing the interface `BallotChangeListener`. Another option is to be notified of changes in the total vote of an `ElementCorrespondence`. To receive these events, a refiner has to implement the interface `ECChangeListener`. The total vote can change because refiners might vote for `ElementCorrespondences` upon reception of an event.

Refiners are associated with plugins. The mapper will obtain them from the plugins after all matchers have been run. In the next step each refiner is given the chance to make one initial sweep over the schemas without any events occurring. This is mainly done for performance reasons but is also useful to allow the refiners to find correspondences they already want to work on. After this all refiners that implement `BallotChangeListeners` are registered with the `Ballot` instance used in the

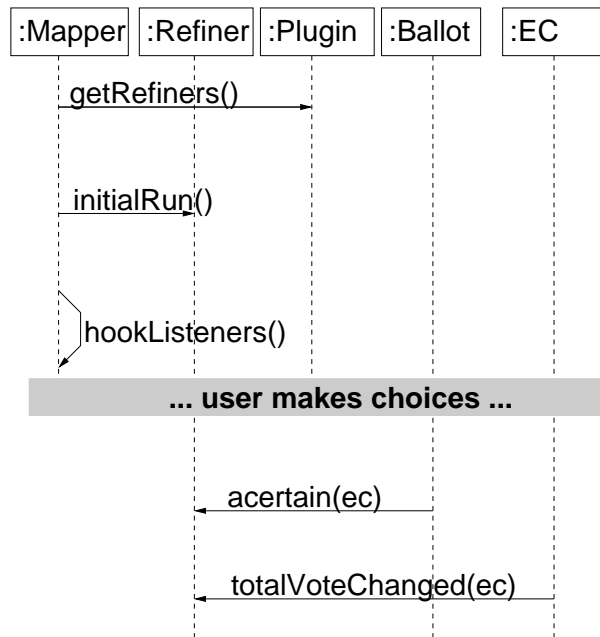


Figure 4.5: Sequence diagram of refiner invocation.

mapping process and all implementors of `ECChangeListener` are registered with all `ElementCorrespondences` from the ballot. After this, refiners will be invoked by the events they have subscribed to.

4.4.2 Impossible Correspondences

If an element from the source schema is already mapped onto a certain element in the target schema, it is assumed that it cannot at the same time be mapped onto another element from the target schema. Considering that this also works in the opposite direction allows the system to drop some suggestions every time they make a choice. This is a somewhat limiting assumption. The majority of correspondences works this way. The ones that do not are usually rather complex and not discoverable by automatic means anyway. In practice this strategy is a good compromise between aiding the user by reducing the length of the suggestion list and losing some functionality in rare cases (then mappings will then have to be specified manually).

At first glance this seems to be implementable in a refiner that uses the `BallotChangeListener` interface. However, there is a catch: the mapper does not guarantee any order the refiners will be run in (after all that order depends on the algorithm and storage structure used in the `Ballot` class to fire the event). It is important to eliminate the impossible correspondences before running any other refiners to avoid lots of chain reactions that try to work on correspondences that should not exist any more and may even cause changes to other (still valid) correspondences in the process. These changes should never have happened because they are caused by correspondences that should not have been there.

As a consequence of this, the functionality to remove impossible correspondences is not implemented as a refiner but in `Ballot` directly. Every time the user marks a correspondence as correct the ballot will first delete all the now impossible correspondences before firing the `Ascertain` event.

4.4.3 Auto Resolver

When going through the suggested correspondences one often finds many that are extremely obviously a match. This is especially true for very similar schemas. There may be two suggestions for an element, one with a total vote of about 5.0 and a lot of votes, while the other one has a total vote of around 1 and few votes. This leads to the idea that the system should be extended to automatically resolve correspondences in these cases.

This idea is implemented in the refiner called `AutoResolver`. This refiner can be configured by four parameters that describe the conditions to be met by an `ElementCorrespondence` to be automatically resolved:

1. *Active*: It can be turned off altogether. Default is *on*.
2. *MinTotalVote*: The minimum total vote a correspondence has to have received to be automatically resolved. Default is 2.8.
3. *MinMatchers*: the minimum number of matchers that need to have voted for it. Default is 3.
4. *MinVoteDifference*: This is a relative measure. The correspondence *c1* to be automatically resolved has to meet the following condition with every correspondence *c2* that has the same source or target elements:

$$c2.totalVote * (1 + minVoteDifference) < c1.totalVote \quad (4.6)$$

The default value for `minVoteDifference` is 0.3.

See the chapter on options 4.8 on how to change these values.

The `Auto Resolver` implements the `ECChangeListener` interface. It iterates over all `ElementCorrespondences` on its initial run. It resolves all correspondences that meet the above criteria. Afterwards it reacts to events from the `ElementCorrespondences`. Every time the total vote of a correspondence has changed, it will be reevaluated. If it matches the criteria with the new vote it is resolved.

4.4.4 Children Refiner

Once the user has marked a correspondence as correct, correspondences between the sub-elements of source and target become more likely.

The `Children Refiner` implements `BallotChangeListener` to be notified every time the user marks a correspondence as correct. It will only take action on `ElementCorrespondences` that have single source and target elements. The `Children Refiner` then iterates over all possible pairs of sub-elements of the source and target elements. If it can find an existing `ElementCorrespondence` instance for such a pair, it votes for it. Otherwise it does nothing.

Voting for a correspondence can cause the `Auto Resolver` to resolve it. This may lead to a (wanted) chain reaction. As a result of this, the user should first take care of the root element of a schema. He could be helped by the system in doing so, see chapter 6.2 on why this is a bit difficult.

4.5 Plugins

As outlined in chapters 2.3.4 and 3.8 there may be a need to extend the system with both domain knowledge and matching algorithms that implement either domain knowledge or new approaches to finding matches.

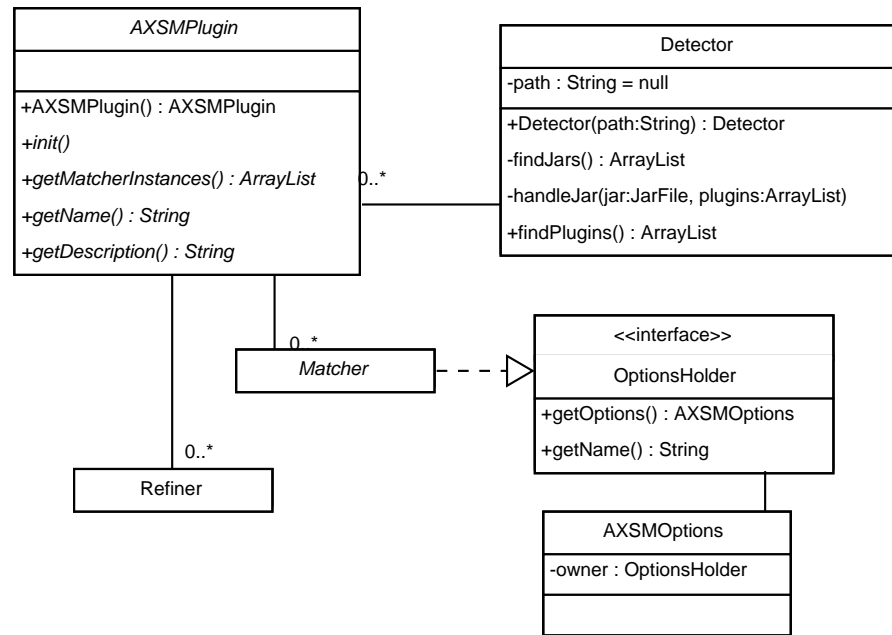


Figure 4.6: Class diagram of the plugin infrastructure

To simplify the development and deployment of these extension AXSM has a plugin interface. This interface allows the implementation of new matchers and refiners without a complete understanding of the whole system architecture. It also allows matchers and refiners to be associated with a plugin. Plugins are contained in JAR files and deployed by simply copying these to the plugin directory of AXSM. See figure 4.6 for an overview of the relationships between plugins, matchers, refiners and detector.

When the mapper is run, it will first employ an instance of the Detector class to search all the JAR file in the plugin directory for subclasses of `axsm.plugin.AXSMPlugin`. These are then instantiated and used to obtain the matchers and refiners.

Redetecting plugins on every run of the matcher may not be very efficient but makes deployment as well as incorporating AXSM into another application easier.

Appendix D explains the development of plugins step by step.

4.6 User Interface

The user interface of this prototype is not meant to be a fully-featured environment for everyday use. It serves as a demonstration of what AXSM can do and eases testing of new matchers quite a bit. In a production system that uses AXSM as a component (see appendix E on how to do this) that system's specific interface will be used anyway. However, it is important to have an application that gives a concrete example of how to handle the component.

To keep the development effort low, the user interface is a web-application using Apache Tomcat as a web-container and Struts [8] as a framework for the web-application. Figure 4.7 shows the page listing all the matching candidates. For each



Figure 4.7: Trivial example: Screenshots. Provide schemas verbatim (a) or as URL (b), suggestions displayed (c) and after marking rectangle-box as correct (d), the final output (e).

element from the target schema, the possible match candidates from the source schema are shown. For each candidate the suggested rule and a detailed listing of votes are given. The user can interact with the system by using the links on the right hand side of the page. He can indicate that a suggested match is correct (including the rule proposed), that the elements are right but the rule is wrong, or none of the given candidates actually match the element. Every time the user makes a choice on an element, the list of candidates is updated. While the list is usually overwhelming at first, its length quickly reduces as you make a few choices thanks to the refinement feature.

Important to the user is the point of view the data is presented from. You can either look at what source element could go to which target elements or where the data of the target element comes from. At first glance there does not seem to be much of a difference. There is however, because the candidate-relation between source and target elements is an n:m relation in general and each individual element corresponds to multiple ones in the other schema.

To the opinion of the author, it is often easier to look at things from the target perspective asking the question "What do I need to do to fill this element?". However, the source perspective sometimes is useful and different people will have different tastes, thus AXSM's classes support both perspectives and the user interface can show them as well⁴. The presented frontend also lets the user choose which perspective suits the given situation best.

Especially with large schemas it is easy to get confused in either perspective with elements of similar or equal names. The context of an element is also not always obvious. To find out more about an element, the user can just click its name and an information page about this element is shown.

Once the user made all the choices he wants to make (he may leave some open), he can click finish and a page with the results in a XML representation is displayed. There is no need to copy-and-paste this XML code, it is also available for download as a single file. The format used is discussed in the next section.

Please also note that security was not paid attention to at all in the user interface. It is to be considered insecure for use in an unsafe environment.

4.7 Output Format

4.7.1 Why not XSLT?

A common format to store transformation rules for XML documents is XSLT. It is convenient for storing a complete transformation as this transformation can then be quite easily applied to XML instance documents using standard software. Unfortunately, XSLT has some drawbacks that make it ill-suited as an output format for AXSM:

- The level of abstraction is rather low. This is a problem when the specified mapping is to serve as an input for further tools such as visual mappers. These mappers would be faced with the task of parsing the XSLT to find correspondences between elements. A rather difficult task as the representation XSLT uses is far away from how AXSM represents correspondences and also any other likely representation format.

⁴Grouping the candidates by source or target element is quite an ugly task. It will unfortunately be a common one in systems that use AXSM. The class `Ballot` greatly simplifies it by providing methods that return the candidates grouped by source or target.

- There is no way of storing match candidates (there are multiple per element) in a sensible way in XSLT. Thus it is only suitable for as the final output format of completed mappings.
- Loss of information. Besides not being able to store match candidates, one can also not store the rule proposed explicitly. The rule will of course be reflected by the generated XSLT code, but getting back an abstract notion of copying or converting from that code is very difficult.

4.7.2 Own Schema

As detailed above, a higher-level abstraction than XSLT was needed. As no suitable existing format could be found, a new one was created. It is an XML language that allows element correspondences to be stored in a straightforward manner. On each correspondence the source and target elements are stored as well as the rule to be used to map them. The XML Schema describing the language in detail can be found in appendix B.

This schema makes a difference between candidates and correspondences, where the correspondences are what the user has already confirmed and the candidates are the suggestions of the system. Each element (from both source and target schema) can either appear in one correspondence or in multiple candidates (it can also not appear at all, as is the case if AXSM could not find any candidate for it).

Figure 4.8 shows an UML diagram of the schema. It very closely resembles the class diagram of the correspondence datastructure in figure 3.4. The major difference is that the entry point is called "mapping" in the schema and "Ballot" in the Java classes. Just like the ballot, the mapping holds to collections of correspondences, one ascertained the other one suggested. Each correspondence has child elements storing source and target elements, the rule used in mapping as well as the votes cast.

4.8 Options

Many matching algorithms have parameters that the user might want to change depending on the problem faced. In an effort to keep the AXSM framework decoupled from the presentation logic, the matchers know nothing about rendering their options for some kind of user interface. Instead, all parameters of a matcher are contained in a JavaBean. To indicate that the matcher is configurable the standard way, it implements the OptionsHolder interface. The mapper can then obtain the options Bean of the matcher by invoking `getOptions()`. This will in the general case return a matcher-specific subclass of `AXSMOptions` that has properties which control the matchers behavior. Refiners can also use the auto detection mechanism, but are not forced to provide options by their superclass.

Frontends can use these JavaBeans to render a user interface that allows adjusting of the options. An example of this can be found in the web-application that is presented as a user interface as part of this project (see the package `axsm.frontend.options.*`). If frontends implement the options functionality using Java's reflection mechanism it is possible to generate the user interface at runtime with little knowledge about the options. This allows the frontend to automatically adapt to new plugins without requiring any changes in the frontend's code⁵.

In an effort to keep the task of handling the options simple, the JavaBean classes containing the options are a little restricted:

⁵As achieving this was not the main focus of this project, it is discussed in further detail in the documentation on the enclosed CD (topic "Options").

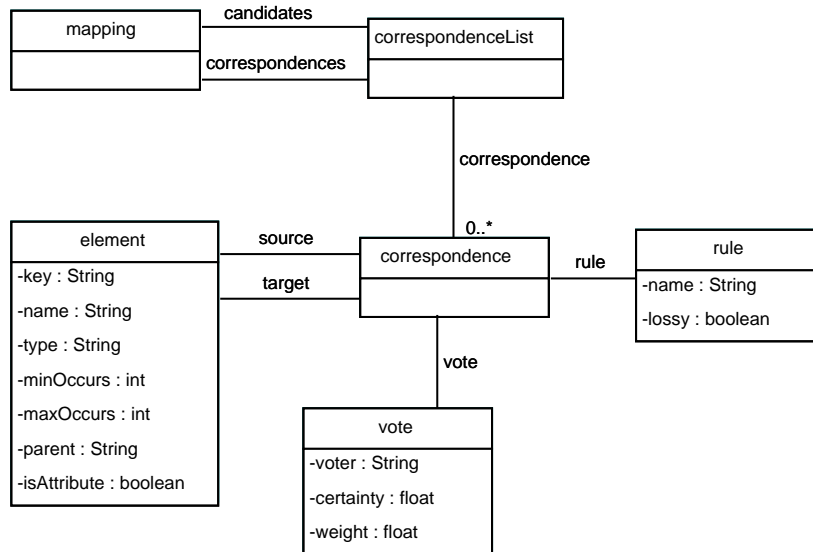


Figure 4.8: Output schema. Everything shown here as attribute is actually implemented as elements (with the exception of `isAttribute` in `element`). An element "mapping" of type `mapping` is the root element

- Only atomic properties are allowed. (This might be worth extending.)
- Properties have to be of the Java types `String`, `boolean`, `int`, `double` or `float`.
- Properties have to be accessible via `getX` and `setX` methods (where `X` is the property's name). Note that this also has to hold for properties of type `boolean` whose accessor methods are often prefixed "is" instead of "get".

Chapter 5

Evaluation

5.1 Testing

5.1.1 General Note

The presented prototype assists the user in creating a mapping. As was anticipated in chapter 1 it cannot create the complete mapping itself. To find out, how good the proposed matches are and what major problems the system has, a number of tests was conducted. These cases are discussed in this chapter.

Most test cases are artificial, because it is suprisingly hard to find some from the real world. There seem to be two major reasons for this. One is that development of most applications had already started long before XML Schema became popular. The other that many schemas used in import or export business processes – where conversion problems often occur – are written for the companies involved and are not publically available.

5.1.2 Statistics

On each of the test cases in the next section you will find a statistics box. The following numbers are given:

1. *Total number of candidates*: Total number of all matching suggestions (candidates) the system has generated.
2. *Covered elements without correct mapping*: Elements from either source or target that have suggestions, but none of the suggestions are correct. This is disturbing to the user as it requires consideration of many options without any benefit.
3. *Source elements covered*: All source elements for which the correspondence with the highest certainty is the correct one. This means that processing the set of suggestions for this element just involves marking the first one as correct.
4. *target elements covered*: The same for target elements. Note that this number can only be different from the source number if multi-element rules are involved.
5. *Source elements covered manually*: The number of source elements covered by the largest possible mapping that is achievable using manual tools.
6. *Target elements covered manually*: The same for the target side.

The overall number of elements can be larger than one might expect in a certain XML Schema. This is because each use of the an element is taken to be an independent element in AXSM's representation of the schema. If an element is declared globally and referenced several times, each of the references as well as the original declaration are treated as individual elements.

This is also the reason why some schemas have manual mappings well below 100% even if instance documents are completely mappable without any loss of data. The difficulty for the matching algorithms is that you cannot tell whether a globally declared element is declared as such just because it needs to be referenced or whether it is acutally a potential root element.

5.2 Simple Cases

5.2.1 Person Lists

Figure 5.1 shows two simple schemas that specify the same XML structure in different ways. The mapping is quite obvious to a human. However, even this simple example illustrates some of the difficulties faced when trying to carry out an automatic matching process:

- Complex types can be named and declared globally (as in schema 1) or can be declared locally and anonymously inside the declaration of the element that is of the type. The same applies to elements: they can be declared globally and referenced (not used in the examples here) or locally inside a complex type.
- There can be multiple elements of the same name in different locations. schema 1 has two elements named "firstname" and in this case it is quite obvious (even to the computer as will be discussed later) which of them maps to the "firstname" element in schema 2. However, the relation is not always this obvious resulting many suggested correspondences with the same total vote. This can be observed later (i.e. in the Bibtex XML case in chapter 5.3). The means to handle this issue will be discussed there as well.

In this simple case, the matcher does pretty good. Out of the suggested correspondences for each element the one with the highest vote is always the correct on. Choosing the root element of schema 1 "personList" to correspond to the root element of schema 2 "personList2" will result in an automatic resolution of all elements but the globally declared "firstname" in schema 1. Not mapping the global "firstname" in schema 1 is correct as there is not partner for it in schema 2.

Note that the system is able to tell the difference between the two elements called "firstname" in schema one with the help of the Children Refiner (see chapter 4.4.4) that will vote for the sub-element of both "person" elements to correspond.

5.2.2 Invoice Schemas

As a second artificial test case, two schemas representing simple invoices were constructed. The invoices will hold the same data but there are some differences in the representation:

- Each have order lines but in one the total price of the line is stored (= *itemPrice*quantity*) while in the other one the price of one item is stored.
- The addresses of sender and receiver are stored differently. There is an additional level of nesting in one schema and the names of the companies are stored at different levels.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:s1="http://localhost/schema1"
  targetNamespace="http://localhost/schema1"
  elementFormDefault="qualified" >
  <xs:element name="firstname" type="xs:string" />
  <xs:element name="shoesize" type="xs:integer" />
  <xs:complexType name="personType" >
    <xs:sequence>
      <xs:element ref="s3:firstname" />
      <xs:element ref="s3:shoesize" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="person" type="personType" />
  <xs:complexType name="personListType" >
    <xs:sequence>
      <xs:element ref="s3:person" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="personList" type="personListType" />
</xs:schema>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:s2="http://localhost/schema2"
  targetNamespace="http://localhost/schema2"
  elementFormDefault="qualified" >
  <xs:element name="personList2" >
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded" >
        <xs:element name="person" >
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="shoesize" type="xs:integer" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Figure 5.1: Simple person list schemas.

<i>schema3.xsd to schema5.xsd</i>	absolute	relative
Total Number of candidates	16	-
Covered elements without correct mapping	0	0
Source elements covered	4	0.8
Target elements covered	4	0.57
Source elements covered manually	4	0.8
Target elements covered manually	4	0.57

Figure 5.2: Statistics of the simple person list case.

As an illustrative example instances complying to each schema are given in figure 5.3. The XML Schemas for these would not fit on one page together making comparing them difficult on paper. The enclosed CD has both schemas (`testdata/l2/*.xsd`) and a HTML page that shows both side by side (`testdata/l2/compare.html`).

Theses two schemas are less similar than the ones from the first example. The difference that has the greatest impact on the automatic matching process is the different expression of the addresses and names of sender and receiver. The order lines are correctly treated by the mapper and marking bill-invoice and orderLine-line as correct automatically resolves the remaining elements in this branch. However, the names of the companies in the target schema would have to be pulled out of the "sender" and "receipient" elements in the source, while the other part of these elements can be almost directly copied into the address records on the target side. As there is no matcher that tries to assemble records using elements from different hierarchy-levels, this cannot be handled automatically. The rest of the lower-level elements ("country" and "city") in this branch are correctly handled by the Auto Resolver. Most others have suggested correpondences that will allow easy manual mapping. However, the system is at loss of how to map the intermediate level elements "to", "from" and "sender", "receipient", "company" correctly. This will have to be sorted out using a front end for manual specification of mapping rules.

5.3 Different Bibtex XML formats

To different approaches that specify Bibtex records in XML Schema were used as a test with real schemas. The XML Schemas come from two similar projects located at <http://bibtexml.sourceforge.net/> and <http://www.authopilot.com/xml/> respectively. As the schemas are very long (400 and 1000 lines) they are not printed here but can be found on the CD. "nyberg/schema.xsd" was used as the source schema, "bibtexml/bibtexml.xsd" as the target schema in this test.

The two schemas are very similar, especially in the choice of element names. This makes it rather easy for the mapper to find mapping candidates. There is one big difference: the source schema has an element called "nonStandardField" that can store name-value pairs that do not belong to the basic set of fields for each entry (i.e. ISBN numbers for books are stored this way). The target schema on the other hand does not make use of such a generic way to store values. It instead has some more specific elements that hold these values. This difference appears in a high number of places (on each type of publication) but is always exactly the same.

There are also some minor differences in representation between both schemas and a complete mapping is not possible. One example of this is that the editors of a publication are stored as a flat string in the target schema but in a nested way (with each name having its own element) in the source. As there is no matcher that automatically detects lists, this as to be sorted out manually. One can mark corre-

```

<b:bill xmlns:b="http://localhost/l2/source.xsd">
  <b:addressing>
    <b:sender>
      <b:company>Straw Importers Inc.</b:company>
      <b:street>Farm Rd. 42</b:street>
      <b:city>12345 Anywhere</b:city>
      <b:country>A</b:country>
    </b:sender>
    <b:recipient>
      <b:company>The Farm Shop Ltd.</b:company>
      <b:street>Nowhere Rd. 10</b:street>
      <b:city>12347 Anywhere</b:city>
      <b:country>A</b:country>
    </b:recipient>
  </b:addressing>
  <b:line>
    <b:qty>50</b:qty>
    <b:name>Original Straw Grade A</b:name>
    <b:itemPrice>20.74</b:itemPrice>
  </b:line>
</b:bill>

<i:invoice xmlns:i="http://localhost/l2/target.xsd">
  <i:from>
    <i:name>Straw Importers Inc.</i:name>
    <i:address>
      <i:street1>Farm Rd. 42</i:street1>
      <i:street2></i:street2>
      <i:zip>12345</i:zip>
      <i:city>Anywhere</i:city>
      <i:country>A</i:country>
    </i:address>
  </i:from>
  <i:to>
    <i:name>The Farm Shop Ltd.</i:name>
    <i:address>
      <i:street1>Nowhere Rd. 10</i:street1>
      <i:street2></i:street2>
      <i:zip>12347</i:zip>
      <i:city>Anywhere</i:city>
      <i:country>A</i:country>
    </i:address>
  </i:to>
  <i:orderLine>
    <i:quantity>50</i:quantity>
    <i:name>Original Straw Grade A</i:name>
    <i:totalPrice>1037</i:totalPrice>
  </i:orderLine>
</i:invoice>

```

Figure 5.3: The same invoice expressed using different schemas.

<i>source.xsd to target.xsd</i>	absolute	relative
Total Number of candidates	73	-
Covered elements without correct mapping	3	-
Source elements covered	9	0.64
Target elements covered	9	0.5
Source elements covered manually	11	0.78
Target elements covered manually	12	0.67

Figure 5.4: Statistics of the invoice case.

<i>nyberg.xsd to bibtexml.xsd</i>	absolute	relative
Total Number of candidates	5689	-
Source elements covered	72	0.8
Target elements covered	72	0.36
Source elements covered manually (appx ^a)	430	1
Target elements covered manually (appx)	190	1

Figure 5.5: Statistics of the Bibtexml case.

^aThe numbers for a manual mapping are approximate because the author is not quite sure about the semantics of a few of the elements.

spondences between "editors" from the source and "editor" from the target schema as being essentially correct but having the wrong rule. This correspondence can later be clarified using manual mapping tools.

As the schemas deal with different types of publications, many element names occur multiple times in different places (i.e. "address" can be found in 11 places). This leads to a high number of match candidates (5689) and an overwhelming list presented to the user. Marking the top-level elements "entry" to correspond allows the refiners (see chapter 4.4) to differentiate between all the elements with identical names but different parents. This brings the list of candidates to a slightly more comfortable length of 2327 with over 59% of the source elements mapped. Almost all of the remaining candidates are caused by the nonstandard field issue discussed above.

Apart from failing to map the representations of nonstandard fields, the system performs quite well in this case. Being able to match 59% of the source elements with just one helpful mouse click of the user is certainly satisfactory. This also shows that it is very important for the user to consider the order in which the candidates are processed. A wrong ordering (leaf elements first) can cause a lot of unnecessary work, while a smart ordering (the root element first) can save quite a bit. Also see chapter 6.2 on why the system does not show the candidates for the root element first.

5.4 Auction Search Results

Because of the difficulty to obtain more real world test data that was suitable for this system a test case was invented that should be close to reality. To get some realistic¹ schemas two persons wrote schemas that represent approximately the same data. However, the specification of what was to be stored in instance documents complying to these schemas was very brief. The task was very imprecisely worded as "write a schema that represents the results of a search at e-bay". The two schemas are a bit

¹Realistic i.e. means that they are sufficiently different in naming as well as use of types.

<i>auction.xsd to ebay.xsd</i>	absolute	relative
Total Number of candidates	63	-
Covered elements without correct mapping	8	-
Source elements covered	8	0.57
Target elements covered	8	0.4
Source elements covered manually	10	0.9
Target elements covered manually	11	0.5

Figure 5.6: Statistics on the auction search schemas. The manual mapping has more mapped elements on the target side, because the target schema stores some redundant data (number of bids).

long to print but can be found on the enclosed CD.

As expected, there are differences between the schemas. Some points are more modeled in greater detail in the source schema *auction.xsd*, while others are richer in the target schema *ebay.xsd*. There are some noteworthy differences:

- *auction.xsd* uses global element and type declarations while *ebay.xsd* nestes everything and therefore uses anonymous complex types. This is usually not a problem in the mapping and was already discussed earlier.
- In *auction.xsd* monetary amounts are modeled as elements of primitive type (*xs:decimal*), while *ebay.xsd* uses a complex type that also stores the currency. This is not mappable manually. AXSM was able to match the amounts by name, but this is not really correct without assuming some currency on the target side.
- A few elements appear in one but not the other (that is why the best manual mapping is only 10 and 11 elements, respectively). This discrepancy in elements rises from the fact that *ebay.xsd* stores some redundant data (the number of bids) that can be computed from *auction.xsd* (the length of the bid list).
- The source schema *auction.xsd* restricts the domain ranges of some simple types which *ebay.xsd* does not.

Matching 9 elements where a manual match of 10 or 11 is possible is a good result. The auction schemas also show another problem of schema mapping: Slight differences in semantics of the elements might lead to very complicated mappings. An example of this is the way bid values are stored here. If one side just assumes one currency (Euro) and the other specifies potentially any currency, you have to give a complete table of all exchange rates. This would also have to be dynamic to find the current rate at any time two instance documents are mapped.

5.5 Usablilty of the Frontend

The frontend was chosen to be implemented as a web-application because this saved development time. Major disadvantages of web-applications are the very limited set of UI elements and noticeably longer response times when compared to thick client frontends.

On small schemas the web fontend works well. The list of candidates is not very long and display of new pages is reasonably fast. As the schemas get bigger, the suggestion list becomes longer and loading as well as rendering times increase. What is still considered usable here depends on the computer used and also on the patience

of the user. However, the suggestion list for the bibtexml schemas (about 5700 entries) was long enough to frustrate the user.

The chosen presentation layout is appropriate if there are not very many suggestions per element. The per-element list should fit on one or two browser screens to avoid getting lost in it. There are also some issues with table formatting turning ugly if the window is too narrow, but this is dependent on the browser used and the specific setup.

In general the formatting possibilities in HTML are somewhat limited and a much better frontend would be possible if HTML were more powerful. However, this would take much longer to implement and was not possible during this project. Representing correspondences graphically (i.e. by drawing lines between elements) would certainly upgrade the mapping experience of the user.

The error messages by the frontend are not very helpful. Especially the error one gets, if one of the schemas could not be read. It is just a generic error that gives next to no information of what actually went wrong (file not found, some kind of XML error etc.). As this can be quite frustrating, users are advised to test the schemas in other application. This is a big issue and has to be resolved in a system that is meant for any kind of productive work.

5.6 Complexity

The algorithmic complexity of the overall mapping system is mainly influenced by the matchers. Assuming that each matcher looks at every pair of source and target elements (which is true for most matchers), one sees that the complexity of each matcher is dependent on the sizes of source and target schemas. More specifically the complexity under these assumptions for each matcher is:

$$C_{\text{Matcher}} = O(N_{\text{source}}N_{\text{target}}) \quad (5.1)$$

Where N_{source} and N_{target} are the number of elements in source and target schema, respectively. This complexity measure serves as a lower bound for most matchers. Some matchers do even worse, because they go over the schema multiply times (i.e. the Type Signature Matcher). To do better than this, a matching algorithm must have some way of finding all matches without looking at all elements individually. To do this, support by a special access path in the schemas is usually needed (i.e. by element name if the matcher looks for exact name matches). However, providing this access path just shifts the work from the matcher to the SchemaContainer. This only reduces the total cost, if this access path is used by at least two matchers.

Looking at the complete Mapper that will run M matchers individually the complexity is (using equation 5.1):

$$C_{\text{Mapper}} = O(MN_{\text{source}}N_{\text{target}}) \quad (5.2)$$

Any effort to reduce this total level of complexity would need to either reduce the level of *all* individual matcher (as discussed above) or do away with the architectural concept of the composite mapper. Tightly integrating all matchers results in a hybrid mapper that might have a lesser complexity level. However, this comes at the cost of losing all the benefits of the composite mapper (mainly extensibility and flexibility at runtime).

An option that does not reduce the level of complexity but might still significantly cut run time would be to have an algorithm that decides whether to run a matcher. If this algorithm's complexity is of a lesser level than $O(N_{\text{source}}N_{\text{target}})$ time is saved

if the matcher is not run. This only makes sense on matchers that are rather likely not to be run (i.e. domain-specific ones). Note that the overall level of complexity still remains the same, as the matchers that are run are still $O(N_{source}N_{target})$ and M in equation 5.2 is only reduced by a linear factor.

5.7 Summary

The above testing shows both strengths and weaknesses of AXSM. Mapping schemas that are very similar in both naming and structure is obviously an easy task to the system (see first example). Even if the naming of the elements is rather different a mapping can still be achieved thanks to the Synonym Matcher. This is of course only the case if entities in one schema describe concepts that can also be found in the other schema.

A mismatch between conceptual entities can yield problems. A very common mismatch seems to be the representation of lists. On one side they might be represented in one element (i.e. with comma-separated entries, nothing in the XML reflecting it is a list), the other side might make the list explicit by having a list element that encapsulates a collection of elements representing one entry each. Note that in general a mapping between these representation is only possible in one direction, the other direction requires knowledge about the format used in the list.

It seems possible that additional matchers would be able to resolve some structural problems. However, many of these problems are one-way (like width times height equaling area) and the use of these domain specific matchers is a little limited. Also see the future work section in chapter 6.2.

Overall the system greatly reduces the work involved in specifying a mapping. However, every suggestion still needs human checking. Just taking the correspondence with the highest certainty for every element would give a mapping that is correct in many places, but has some serious wrong matches. This is especially true if the overlap between two schemas is not very big as the matchers will give their best guess for a correspondence on each element, even if it is not matchable.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

6.1.1 Architecture

As discussed in chapter 3 there are two different architectural styles for mapping systems. The one chosen here is the composite approach. This choice has turned out to be a good one. During testing it became clear that the ability to extend the system with additional domain knowledge and mapping algorithms is more important than the performance increase that might have been possible in a hybrid architecture. As a side note the individual mapping algorithms are already complex enough to make maintaining some of them a difficult task. If these algorithms were incorporated into one, this would become even more complicated.

Another important aspect of AXSM's architecture is the facilitated integration into mapping systems as a component. The tests described in chapter 5 have shown that it is not possible to achieve complete mappings with the current state of the system. The ability to further process the generated mapping suggestions transparently for the user is therefore very important. The author believes that integration with other systems or implementation of a better visual frontend are tasks that do not require a great effort as far as the interface with AXSM is concerned. More documentation on this can be found in appendix E.

One major advantage of the composite system architecture is the ability to incorporate user extension without having to change the system's code. This has many benefits, independent implementation and simplified testing come to mind as examples. The choice of Java as an implementation language was very helpful in implementing this through the language's reflection capabilities. As a demonstration of what the plugin interface is capable of, the matching algorithms that are build into¹ the system are implemented as plugins. The interface provides a both simple and powerful way that allows developers to write further extensions without having to understand the whole system let alone touch the system's code. The plugin interface is described in greater detail in chapter 4.5 and appendix D.

An important extension to the architecture is support for different kinds of visualizations. This could be done using plugins in a similar way plugins are handled now. The main flow of control would then stay with AXSM and the visualization plugin could implement hook methods in different places. Note that this approach probably would not work very well with a webapplication frontend. In that case there would

¹"Build in" should rather be "delivered with" as the algorithms can be seperated from the system very easily.

be two frameworks (the webcontainer and AXSM) competing to manage the control flow.

6.1.2 Technologies

The choice of XML Schema as an input format seemed obvious. However, when looking for examples of mappable schemas for real application, one realizes that many systems are older than XML Schema itself and thus still use DTDs. An example for this are different XML formats for office suits. This situation is expected to improve as XML Schema is adopted by more and more applications.

Other than this difficulty of finding examples, XML Schema was a good choice. It provides a lot of information and is likely to be used as a format specification language. DTDs might still be a bit more popular, but for mapping purposes they are not very useful as described earlier.

Java as implementation language was also well suited. The reflection mechanism was helpful in creating the plugin functionality as well as in handling the options. Using a language that is based in a virtual machine was also important to have portable plugins. This would have been very difficult in earlier generation languages such as C++ or Pascal.

6.1.3 Algorithms

There are also some points about the individual algorithms that are worth mentioning:

- The initially planned Exact Name Matcher was dropped because it just casts the same votes as the PartialNameMatcher.
- The LevenshteinMatcher does not adapt very well to different length of element names. There is some basic support for varying length, but it might be necessary to reconsider whether this should really be linear. There is also some danger of Levenshtein Matcher and Partial Name Matcher casting same votes because you get part of a name by deleting characters.
- The Type Signature Matcher does a good job at figuring out compatible types in cases where no name correspondences can be found. This works especially well in structurally rich schemas, that use many different atomic types.
- The Same Type Matcher's capabilities to detect types the source type could be converted to should be extended. This quickly becomes very complicated as one also has to look at domain ranges and XML Schema extension/restriction hierarchies. In the process it should be renamed to "Compatible Type Matcher".

6.1.4 Summary

Considering the current state of the system the application its application will be in assisting humans in creating a mapping. This process is made a lot easier but a complete autonomous mapping is unrealistic at this point.

6.2 Future Work

6.2.1 Enhancements to the Framework

While this implementation already worked well on the tests presented, there is always room to improve. Some ideas that came to mind during the project are briefly presented here. They were not pursued for lack of time. In addition to these topics writing additional matcher (i.e. SynonymMatchers for languages other than English) is always a worthwhile occupation.

1. *Nested XML particle structures*: These are basically ignored in this implementation as the element tree is flattened (see chapter 4.2). The arguments made for this are reasonable as a first approximation. However, it may be possible to get further information that is helpful to the mapping from the nesting. This was not considered. It should also be noted that there seems to be a trade-off between accuracy of nesting representation and flexibility in matching.
2. *XML Text Nodes*: Text nodes are not supported in complex types.
3. *Matching Speed*: Running every matcher independently is rather slow. It may be possible to find some synergies between datastructures of different matchers. Currently the complexity of the matching process is $O(N_{source}N_{target}M)$, where N are the number of elements of the schemas and M the number of matchers run. Unfortunately, this is unlikely to change without drastic changes to the system. Section 5.6 explains why.
4. XML Schema substitution groups and restrictions are not handled.
5. *Rule Semantics*: Add-on matchers will bring along their own rules if none of built-in ones match their purpose. This can result in different Rule classes with the same semantic in different plugins and thus the matchers each voting for their own. Fixing this might require some form of central registry of rule semantics.
6. *Machine Learning*: Let the system learn by the choices the user makes. Apply the learned in future mappings. This can take different forms and might be implementable as a combination of a Refiner that learns and a Matcher that applies the learned knowledge.
7. *Automatic selection of applicable plugins*: Find out which matchers are worth running. This might greatly reduce costs on large schemas. This could be implemented on plugin or on matcher level.
8. *Document instances*: Also looking at XML instances, instead of only taking the schema definitions into account, will present many new ways to find information for the matching processes. However, it will also involve substantial additions to the framework. Some ideas of what can be done in this area are discussed in [1].

6.2.2 Enhancements to the Frontend

The presented frontend was made for demonstration and testing purposes. It is not suitable for productive use for lack of completeness in features and ease of use. For actual applications there are some important points missing:

1. *Data duplication*: The web-application fronted does not allow the specification of a set ElementCorrespondences that use the value of an element in multiple ways in the other schema (either way). This is a limitation of the frontend. There is nothing in the framework preventing it from handling it. However, the user interface would become more complex.
2. *Multi-element Rules*: The presented frontend cannot render element correspondences that have more than one source or more than one target elements. Again this is not a limitation of the framework but of the frontend. It was accepted as the frontend is only meant as a demonstration application.
3. *Order of Suggestions*: Ordering the suggested element correspondences in such a way that the root elements appear at the top of the list. One should note that there is no sure way of telling what the root element is in an XML Schema if more than one element is declared globally (common practice). An algorithm to make an educated guess is needed here.

In general it seems to be a good idea to have one integrated user interface that supports the whole mapping process starting with the import of the schemas all the way to the final result specifying the complete mapping. One option to achieve this is to use AXSM as a component in an existing system. Another one is extending AXSM to support different visual interfaces much the same way plugins are supported now. This could allow the user to choose the interface that is most appropriate to his current task. Some more extensions to the framework might be needed to handle the generation of the final mapping.

Bibliography

- [1] Erhard Rahm, Philip A. Bernstein, "A survey of approaches to automatic schema mapping", The VLDB Journal 10: 334-350 (2001), Springer Verlag
- [2] Hong Su, Harumi Kuno, Elke A. Rudensteiner, "Automating the Transformation of XML Documents", In Proceedings of the Workshop on Web Information and Data Management, 2001. <http://citeseer.nj.nec.com/su01automating.html>
- [3] Li Yongqiang, "Data Mapping by Using Business Form Copying Metaphor", Master Thesis, The University of Auckland (2003)
- [4] J.C. Grundy, J.G. Hosking, R.W. Amor, W.B. Mugridge, and Y.Li "Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems", Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, USA, 3-6 September, IEEE CS Press, 158-167
- [5] "W3C XML Schema", <http://www.w3.org/XML/Schema>, as accessed on 12 Sep 2003
- [6] "The Castor Project", <http://castor.exolab.org/>, as accessed on 12 Sep 2003
- [7] "WordNet", Princeton University, Cognitive Science Laboratory, <http://www.cogsci.princeton.edu/wn/>, as accessed on 14 Sep 2003
- [8] "Struts", <http://jakarta.apache.org/struts> as access on 22 Sep 2003
- [9] "Java WorldNet Library", <http://sourceforge.net/projects/jwordnet>, as accessed on 12 Oct 2003
- [10] Altova, http://www.altova.com/products_mapforce.html

Appendix A

Acknowledgements

There are a few people I would like to thank for their help and support while I was doing this project:

- John Grundy and John Hosking for supervising this project and providing helpful ideas and directions
- Prof J.W. Schmidt of TU Hamburg-Harburg for supervising on the German side.
- Robert Amor for insightful discussions and pointers to literature.
- Marcus Venzke for pointing me to the Bibtex case.
- Hermann Stoeckle for writing one of the auction schemas.

Appendix B

Schema for XML output

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:axsm="http://localhost/axsm_out_1.0.xsd"
  targetNamespace="http://localhost/axsm_out_1.0.xsd"
  elementFormDefault="qualified" >

  <xs:complexType name="rule" >
    <xs:annotation><xs:documentation>
      Stores one axsm.match.voting.Rule instance
    </xs:documentation></xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="lossy" type="xs:boolean" />
      <xs:any minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="rule" type="axsm:rule" />

  <xs:complexType name="vote" >
    <xs:annotation><xs:documentation>
      Stores one axsm.match.voting.Vote instance.
    </xs:documentation></xs:annotation>
    <xs:sequence>
      <xs:element name="voter" type="xs:string" />
      <xs:element name="certainty" type="xs:decimal" />
      <xs:element name="weight" type="xs:decimal" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="vote" type="axsm:vote" />

  <xs:complexType name="element" >
    <xs:annotation><xs:documentation>
      Stores one axsm.match.schema.Element instance. The key is
      unique schema-wide. parent will only be present, if this element
      was not declared globally in its XML Schema and then reference one
      element it appears in. minOccurs and
```

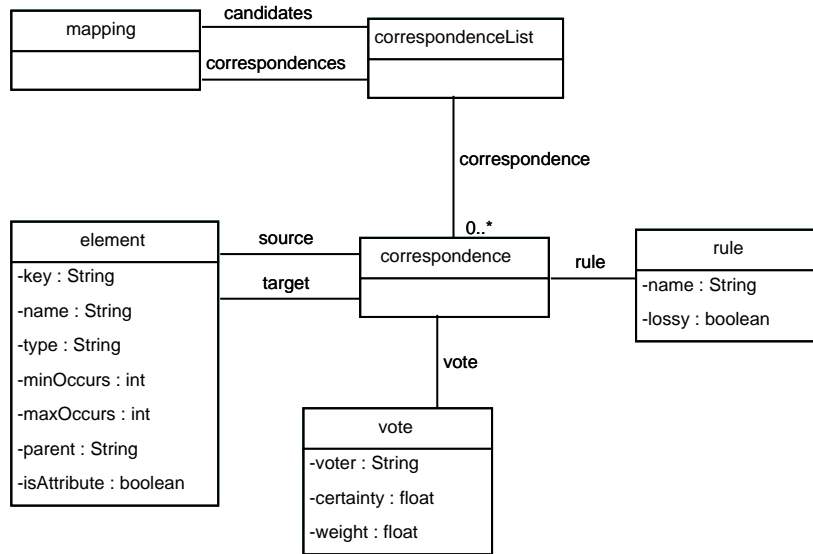


Figure B.1: Output schema. Everthing shown here as attribute is actually implemented as elements (with the exception of isAttribute in element). An element "mapping" of type mapping is the root element

maxOccurs are not the original ones from the XML Schema document because the particle hierarchy is flattened. Instead equivalent cardinalities are reported here.

```
</xs:documentation></xs:annotation>
<xs:sequence>
  <xs:element name="key" type="xs:ID" />
  <xs:element name="name" type="xs:string" />
  <xs:element name="type" type="xs:string" />
  <xs:element name="minOccurs" type="xs:integer" />
  <xs:element name="maxOccurs" type="xs:integer" />
  <xs:element name="parent" type="xs:string" minOccurs="0" maxOccurs="1" />
</xs:sequence>
<xs:attribute name="isAttribute" type="xs:boolean" default="false" use="optional" />
</xs:complexType>
```

```
<xs:element name="source" type="axsm:element" />
<xs:element name="target" type="axsm:element" />
```

```
<xs:complexType name="correspondence">
  <xs:annotation><xs:documentation>
    Stores one axsm.match.voting.ElementCorrespondence instance.
    Note that there can be multiple source or target elements, denoting, that
    these elements are to be used in conjunction by the given rule (i.e.
    concatenate first and last name). There can also be more than one vote,
    but this should be obvious.
  </xs:documentation></xs:annotation>
```

```

    <xs:sequence>
      <xs:element ref="axsm:rule" />
      <xs:element ref="axsm:source" minOccurs="1" maxOccurs="unbounded" />
      <xs:element ref="axsm:target" minOccurs="1" maxOccurs="unbounded" />
      <xs:element ref="axsm:vote" minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="correspondence" type="axsm:correspondence" />

  <xs:complexType name="correspondenceList">
    <xs:annotation><xs:documentation>
      Simply a list of elements of type "correspondence" (either candidates
      or correspondences).
    </xs:documentation></xs:annotation>
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element ref="axsm:correspondence" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="correspondenceList" type="axsm:correspondenceList" />
  <xs:element name="candidateList" type="axsm:correspondenceList" />

  <xs:complexType name="mapping">
    <xs:sequence>
      <xs:element ref="axsm:correspondenceList" minOccurs="0" maxOccurs="1" />
      <xs:element ref="axsm:candidateList" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="mapping" type="axsm:mapping" />

</xs:schema>

```

Appendix C

UML Class-diagram of the Framework

Here is an overview class-diagram of the main framework classes. Most classes are simplified to keep the diagram on one page and readable. Accessor methods are never shown, but one can assume (or read the API documentation on the CD) that they are there for every field.

Appendix D

How to write extensions

This chapter describes how to implement new matchers and plugins to extend AXSM. There is more high-level documentation on the CD. Since this is especially important, it is included in print.

D.1 Framework Architecture

Any matcher provided by a user will be run in a framework-like manner. This means, that you do not use AXSM libraries to write your matcher (that you can call from i.e. your `main(...)` method), but that you write classes that provide pieces of code by overriding methods. These methods will be called at the appropriate times by the framework.

There are mainly two kinds of classes that you will need to write: plugins and matchers. Plugins extend `axsm.plugin.AXSMPlugin`. Their purpose is to collect related matchers and to provide some information about themselves (such as a name and description). Matchers is where the real work happens. Matchers work with the elements of source and target schemas and look for matches between them. They extend the class `axsm.match.matchers.Matcher`.

When you are done developing your extensions, you will usually pack them in one `.jar` file. When you place this file in AXSM's plugin directory, the plugins it contains will automatically be discovered the next time the mapper is run (no need to restart anything). This is very inconvenient for development (see the class `axsm.plugin.ProxyPlugin` and the development section further down on this issue).

D.2 Step by Step

D.2.1 The Plugin

Usually you will begin with a plugin. To create a new one, simply write a class that extends `axsm.plugin.AXSMPlugin`. Your sub-class will be required to implement the `init()` and `getMatcherInstances()` methods as these are abstract. As you can guess, `init()` is called to give the plugin a chance to do any initialization that might be required. The method `getMatcherInstances()` is called to obtain an `ArrayList` of `axsm.match.matchers.Matcher` sub-classes to run. Here is some example code:

```
package somewhere.pack.age;
import java.util.ArrayList;
```

```
import axsm.plugin.AXSMPlugin;

public class MyPlugin extends AXSMPlugin {
    public MyPlugin() { super(); }
    public void init() { //... }
    public ArrayList getMatcherInstances() {
        ArrayList matchers = new ArrayList();
        matchers.add(new SpecialMatcher());
        return matchers;
    }
    public String getName() {
        return "My first plugin";
    }
    public String getDescription() {
        return "Does this or that...";
    }
}
```

You should also implement the `getName()` and `getDescription()` methods.

D.2.2 The Matchers

The matchers is where is real work happens. There can be multiple matchers in one plugin, each serving a specific purpose. Usually, each matcher will implement some sort of rule by which it tries to match elements from the source to elements from the target schema. The idea of the matchers included in AXSM was to keep each one rather small, but to have a lot of them. This means that the individual matcher will be quite simple and thus easy to write and maintain. The logic of combining the results is implemented once in the AXSM framework and you don't need to worry about it. Here is a very basic matcher:

```
package somewhere.pack.age;
import java.util.ArrayList;
import axsm.match.matchers.Matcher;

public class MyMatcher extends Matcher {
    public MyMatcher() { super(); }
    public void findCandidates() {
        // to be implemented later.
    }
}
```

This code leaves a couple of open question. How do you get hold of source and target schema? How are they represented? What do you do when you found a match?

D.2.3 The Schemas

Each matcher has getter and setter methods for both source and target schema. They return instances of `axsm.schema.SchemaContainer` from which you can get the elements by calling `getElements()`. This will give you an `ArrayList` of all elements present in the schema. Each element is represented as an `axsm.schema.Element` instance. Be aware that there can be two different elements of the same name in one schema (if

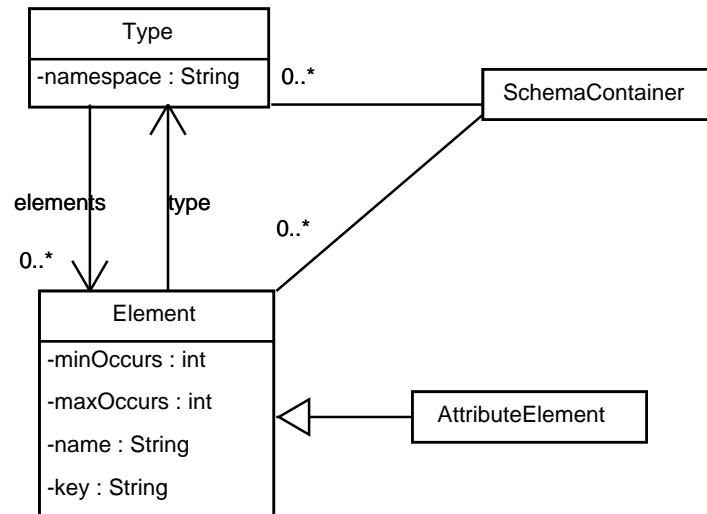


Figure D.1: UML diagram of the simplified schema representation

at least one of the is declared locally). This is why the Elements do not only have a name attribute but also a key. They key is unique within the schema.

D.2.4 Voting

Once your matcher has found a possible match between two elements, it should store this information. It does so by casting a vote for the correspondence between the elements (stored in a `axsm.match.voting.Vote` instance). The votes are collected in a `axsm.match.voting.Ballot`. The ballot stores correspondences between elements as well as votes by matchers for these correspondences. See the UML diagram in figure D.2.

There are a couple of things noteworthy about this class diagram. The Element-Correspondence has `1..*` relations to source and target elements. This means, that a correspondence is not between a single source element and a single target element but a group of source elements and a group of target elements. Some mappings (such as concatenating first and last name to just a name) make this necessary. Each Element-Correspondence has a number of Votes attached to it (cast by different matchers) and also a Rule.

You will learn more about rules later, the essence is that source and target elements correspond using a certain rule (such as just copying the data or converting some measure to a different unit). A matcher thus votes not only for the correspondence of the source to target elements but does so under a certain rule. In other words, the unique key of the ElementCorrespondences stored in the ballot is `(source*, target*, rule)` and not just `(source, target)`.

D.2.5 Rules

Rules specify what must be done to map the source elements onto the target elements. In a lot of cases this is just copying the value of the source element. There are some rules predefined: i.e. `CopyRule`, `ConcatenateRule` and `UnknownRule`. If a matcher

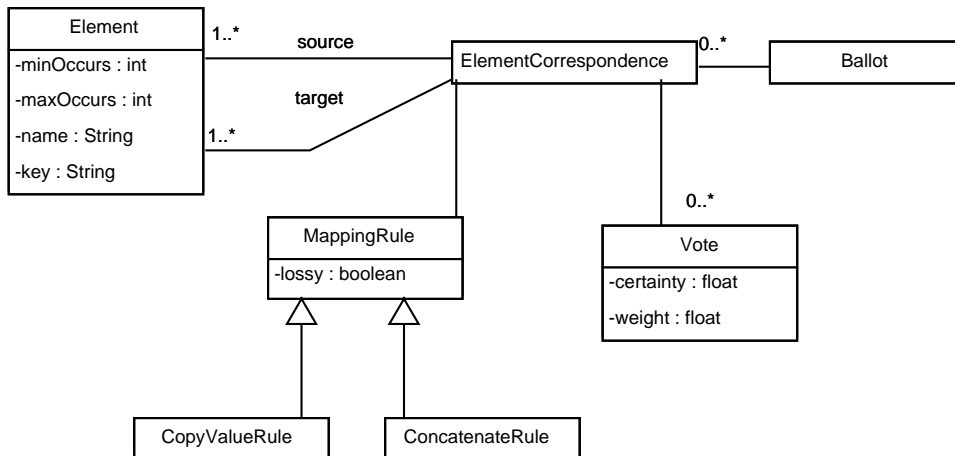


Figure D.2: The correspondence datastructure.

needs its own ones, it can define them by subclassing `axsm.match.rules.MappingRule`. However, one should be aware, that any application that is to work with the results of AXSM must be able to understand the new rule.

The `UnknownRule` is used to indicate that a match between elements has been identified but the system was unable to devise the correct rule. Thus it is usually used to replace an existing rule in the user-feedback step.

D.2.6 Casting Votes

Here is how casting the votes works: Your matcher will be supplied by the framework with a `Ballot` instance. You can use this instance to get `ElementCorrespondences`. The ballot will automatically return existing ones or create a new one if it does not have any matching your specified source and target elements as well as the rule. Here is an example of how this can be done:

```

Element source = // ...
Element target = // ...
ElementCorrespondence correspondence =
    getBallot().getCorrespondence(source, target);
Vote vote = new Vote();
vote.setVoter(this.getClass().getName());
vote.setCertainty(1.0f);
correspondence.addVote(vote);
  
```

How you determine source and target depends on your matcher. The `getCorrespondence(Element, Element)` method assumes, that your rule is `CopyRule`. If you want to supply a different rule use `getCorrespondence(Element, Element, MappingRule)` and instantiate the rule of your choice.

D.2.7 The Refiners

After the matchers have been run, some frontend will usually present the user with the suggestions and let him choose from them. This is also reflected in the ballot. Ob-

viously, the fact that a correspondence is now known to be true can have impacts on the likelihood of other correspondences. These can include that another correspondence has now become impossible or that correspondences between the child-elements of the elements from the correspondence are now more likely than before. Algorithms implementing these and similar things are therefore interested in changes in the ballot.

These algorithms are called refiners here as they are used during the refinement-phase of the mapping. Refiners are implemented as subclasses of the abstract class `axsm.match.refiners.Refiner`. Note that a refiner that merrily subclasses `Refiner` is not notified of any changes to the ballot. To receive change notifications refiners can listen to two events: Acertaining of correspondences and changes in the total-vote of an `ElementCorrespondence`. This is done by implementing the interfaces `BallotChangeListener` and `ECChangeListener`. The framework will take care of registering the refiners with the corresponding instances of `Ballot` and `ElementCorrespondence` respectively.

`Refiner` has an abstract method `initialRun()`. This method is run (on sub-classes) before any Listeners are hooked up. This gives refiners a chance to make an initial (inexpensive) sweep over the ballot to deal with `ElementCorrespondences` that already are in the sought after state. If you don't need this in your refiner, just implement this method empty.

Here is a part of the `ChildrenRefiner` as an example:

```
public class ChildrenRefiner extends Refiner
    implements BallotChangeListener {
public ChildrenRefiner() { super(); }
public void initialRun(){ // do nothing on initial run. }
public void acertained(ElementCorrespondence parentEc) {
    if ((parentEc.getSources().size() == 1) &&
        (parentEc.getTargets().size() == 1)) {
        Element source = parentEc.getSource();
        Element target = parentEc.getTarget();
        Iterator subElementsSource =
            source.getType().getElements().iterator();
        ArrayList voteList = new ArrayList();
        while (subElementsSource.hasNext()) {
            Element subSource = (Element) subElementsSource.next();
            checkSubElement(target, voteList, subSource);
        }
        voteFor(voteList);
    }
}
private void checkSubElement(Element target,
    ArrayList voteList, Element subSource) { //... }
private void voteFor(ArrayList voteList) { //... }
public String getName() { return "Children Refiner"; }
```

D.2.8 Options

Many matchers will have some parameters that have to be adjustable for the user. Every matcher has a weight parameter, that allows the user to change the relative weight of its votes wrt. the other matchers.

AXSM provides means that make offering options to the user very simple. All

you have to do is encapsulate your options in a JavaBean class (that sub-classes `AXSMOptions`) and implement the interface `OptionsHolder` (which is basically just a marker) in your matcher or refiner. The framework will automatically collect the options JavaBeans and offer them to the frontend.

However, there is a catch (to make the rendering for the frontend simpler): The data types of the options are limited to the primitive types `int`, `boolean`, `float`, `double` and `String`. Allowing lists would have made the job for the frontend a lot harder as it has to have generic rendering algorithms for any options JavaBean.

If your `OptionsHolder` is a matcher, a lot of work is already done for you. The class `axsm.match.matchers.Matcher` implements the `OptionsHolder` interface and also handles the instantiation of the JavaBean. All you have to do is override `getNewOptionsInstance()` to return an instance of your options JavaBean instead of the generic `MatcherOptions` (that have just the weight option).

Here is how this is handled in the `LevenshteinNameMatcher`:

```
public class LevenshteinNameMatcher extends NameToNameMatcher {
    // ... lots of other code
    protected LevenshteinOptions getLOptions() {
        return (LevenshteinOptions) getOptions();
    }
    public MatcherOptions getNewOptionsInstance() {
        return new LevenshteinOptions(this);
    }
}
```

Note that the `LevenshteinNameMatcher` also implements a method `getLOptions()` that returns the options JavaBean cast to the correct type. This avoids many casts in the matcher's code and is purely for convenience (you should consider doing this too).

D.3 Tips for Development

Being able to just drop a jar-file with plugins in to a directory and have the plugins discovered automatically is a nice feature for a system that is actually used. It is very annoying to develop plugins this way. You would have to: Write the plugins and matchers, zip them up in a jar-file and copy the jar-file into the plugins directory to run the plugins. This does not even enable debugging.

Fortunately, there is a cure. It comes with the `axsm.plugin.ProxyPlugin` class. This class does just what its name suggests: provide a proxy for a plugin. Here is how you can use this to ease development:

1. For every plugin, create a subclass of `ProxyPlugin` and override the `getPluginInstance()` method to return an instance of your actual plugin.
2. Put just the proxy plugins into a jar-file and copy this file into the plugins directory of AXSM. Do not include any actual plugins or matchers in this jar-file.
3. Develop your plugins with your favourite IDE/editor and keep the compiled classes in a normal directory structure.
4. Put this directory structure on the classpath of our web-container (i.e. Tomcat). If you run the container from your IDE in debug-mode you will now even be

able to put breakpoints in the plugins (not in the proxies, but this should not be necessary as they are extremely simple).

5. To create a "production" jar-file with your plugins, just put all the real plugins and matchers into a jar-file. Do not include the proxy plugins. You can now simply put this jar-file into the plugins directory to run the matchers.

Appendix E

How to use AXSM as a component

E.1 Important Classes

Take a look at the class diagram in appendix C to see how these classes are interconnected. The electronic version of this chapter on the CD has links to API Javadoc pages.

- *axsm.schema.SchemaContainer* Holds the AXSM representation of an XML Schema.
- *axsm.schema.SchemaFactory* Helpful in creating schemas. The static methods allow the user to obtain a SchemaContainer instance by just passing a filename (as a String), Reader or InputStream instance.
- *axsm.Mapper* The main class that handles the automatic part of the mapping process. Simply set the schemas and invoke run() (also see the code example in the next section).
- *axsm.match.voting.Ballot* Collects the results of the matching process. This is where you can retrieve the suggestions and also update them to reflect user input. There are two collections of ElementCorrespondences: Correspondences (the suggestions) and CertainCorrespondences (those that the user marked to be correct).
- *axsm.match.voting.ElementCorrespondence* Represents a correspondence between a set of elements from the source schema and a set of elements from the target schema. Also holds a rule that describes what needs to be done to actually convert the elements.
- *axsm.match.voting.ECContainer* Holds a collection of ElementCorrespondences. Offers various access paths (by key, by source, by target, by source-target-rule).
- *axsm.match.voting.ECIterator* Allows type-safe iteration over a collection of ElementCorrespondences.
- *axsm.schema.Element* Represents an XML schema element.
- *axsm.schema.Type* Represents an XML Schema type (both complex and simple), simplified particle structure, see the javadoc for details.

E.2 Step By Step

E.2.1 Running the Mapper

Running the automatic part of the mapping process is quite simple. You first obtain the XML Schemas that are converted to the AXSM representation by the SchemaFactory class. Next you instantiate a Mapper and tell it about the schemas. Now you can run the mapper by simply invoking `run()`. After that the results can be obtained with `mapper.getBallot()`.

Here is a piece of code:

```
SchemaContainer source = SchemaFactory.getSchema("source.xsd");
SchemaContainer target = SchemaFactory.getSchema("target.xsd");
Mapper mapper = new Mapper();
mapper.setSourceSchema(source);
mapper.setTargetSchema(target);
mapper.run();
Ballot ballot = mapper.getBallot();
```

E.2.2 Getting the Results

The ballot you now have contains all the suggested `ElementCorrespondences`. You will probably want to present them to the user for further refinement. Although you can retrieve the `ElementCorrespondences` just as an `ECContainer`, there are two data structures that will usually be more helpful for use in frontends. The major drawback of the `ECContainer` is, that `ElementCorrespondences` are not sorted at all. It offers access paths by sources or targets, but to use those, you have to know what sources or targets you are looking for. To relieve you of this, the ballot offers means to arrange the `ElementCorrespondences` by source or by target elements. The important methods are `getSortedBySource()` and `getSortedByTarget()`. Both return a `SortedMap`. The keys are `ElementPackages` that hold the source or target elements of the `ElementCorrespondence` together with a integer value that is unique within the `SortedMap`. The values of the `SortedMaps` are `Sets` of `ElementCorrespondences` that have the source or target elements from the key `ElementPackage`.

For an example of how to make use of the nested data structures, take a look at the `displaytarget.jsp` and `displaysource.jsp` pages of the supplied frontend (`mapping-frontend/WEB-INF`).

E.2.3 User Interaction

In the next step of the process, a frontend will usually ask the user to interactively specify which `ElementCorrespondences` are correct. As marking an `ElementCorrespondences` as correct does have implications on the other remaining `ElementCorrespondences` (i.e. some might no longer make sense, others may have become more likely), it is a good idea to tell the ballot about it. AXSM provides mechanisms to make the necessary changes to the ballot (see documentation on refinement for implementation details). The entry point for this is the `Ballot` itself.

When a user marks an `ElementCorrespondence` as correct, use the methods `refineCorrect(...)` to notify the ballot of this. Set the parameter `fireEvent` to "true" if you want the ballot to take further action (i.e. discard impossible `ElementCorrespondences`, it will run all `Refiners`) on this notification.

By invoking `refineNoneCorrect(...)` you can mark a whole collection of `ElementCorrespondences` as not correct. This will not have any further side-effects.

Finally, `refineWrongRule` tells the ballot that the elements from the specified `ElementCorrespondence` do correspond, but that the suggested rule is wrong. The ballot will attach an `UnknownRule` to the `ElementCorrespondence` and then invoke `refineCorrect(...)`.

E.2.4 XML Export

You might want to use the mappings found so far in a different application. To do so it might be a good idea to convert the ballot into an XML file. To do so, use the `XMLGenerator` class.

```
XMLGenerator generator = new XMLGenerator();
Document document = generator.generateXML(ballot);
StringWriter writer = new StringWriter();
try {
    XMLGenerator.serialize(writer, document);
} catch (IOException e) { //... }
```

This will serialize the ballot to the given `Writer` (you could also use a `PrintWriter` instead of the `StringWriter`). The resulting XML file will not be pretty-printed. This is mainly useful of actual storage in a file that no human will ever have to read. This has the advantage of creating a lot smaller files.


If you want a more human readable version use `serialize(document)` to get a `String` that contains pretty-printed XML:


```
XMLGenerator generator = new XMLGenerator();
Document document = generator.generateXML(ballot);
String result = XMLGenerator.serialize(document);
```


Appendix F

The CD

The attached CD contains all the source code written, this report in electronic form, all the test cases, the Javadoc API documentation, and some higher-level documentation of the framework.


 **source** - This folder contains the source code of the framework, the web-application and the plugins

 axsm - core framework

 web - web-frontend


 plugins - Synonym Matcher and Type Signature Matcher

 **documentation** - Complete documentation of AXSM.


 report - electronic version of this report


 presentation - slides of the presentation of this project (HTML)


 javadocs - API Javadocs in HTML format for all the source code

 other - additional documentation on writing extensions and running AXSM


 **testdata**

 auctions - the auctions test case from chapter 5.4.

 bibtex - Bibtexml schemas from chapter 5.3.

 l2 - different format for invoices as in chapter 5.2.2.

 personlist - the simple person list cases in various forms.

 trivial - the trivial example from chapter 1.2.