Automated schema matching for Universal Data Services

Maas, T.

Award date:
2009

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

Automated schema matching for Universal Data Services

By

Tjeerd Maas

Supervisors

ir. R. den Adel *dr. A.T.M. Aerts*

Master Thesis, Eindhoven, April 2009

Abstract

In data warehousing the ETL process is a well known mechanism used for data integration. Data is extracted from sources, transformed to fit the targets needs and loaded to the target. However to get to this stage a transformation model is created by a consultant. This is an intensive and time consuming process. URBIDATA is a company in Eindhoven which also uses this process.

This thesis examines the possibility to automate this process in some way. It researches whether the automated generation of a transformation model could be possible. The main goal is to construct a prototype of a toolkit that can do this and be very flexible and configurable at the same time.

WMgen was designed and tested on a subset of the Westland data models for this purpose. The results show promising results in generating a transformation model. Such a model should still be presented to the consultant first for validation, but takes away a lot of effort.

Preface

This thesis concludes my studies at the Department of Mathematics and Computer Science at Eindhoven University of Technology in the Information Systems area of expertise.

I would like to thank my supervisor at URBIDATA bv, Robin den Adel and Hein Corstens for their tutoring in the area of Geometric data processing and Universal Data Services. Secondly I would like to thank my supervisor at Eindhoven University of Technology, Ad Aerts and the other members of the assessment committee, Geert-Jan Houben and Natalia Sidorova.

Finally I would like to thank my girlfriend Monique, my family and friends for their constant nagging and putting a stick behind the door, but seriously thank you guys for everything. Lastly I would like to apologize to my band for missing some rehearsals. 42.

Tjeerd Maas

TABLE OF CONTENTS

Revisions

Date	Version	Input from	Details
2008-04-08	0.1	T. Maas	Initial version
2008-05-14	0.2	T. Maas	Finished until section 4
2008-06-23	0.3	T. Maas	Section 4 first draft
2008-07-07	0.4	T. Maas	Added sections 1.4 & 1.5. Processed revisions for v0.2. Added descriptions about UDS components (subsections 1.2). Added Weaving Model definition (section 2.2). Added domain descriptions of [0..1] for the algorithms in 3.3.1. Updated all sections in 4, added section 4.2.2.2 about the History Information Object. Added sections 1 through 6.
2008-08-25	0.6	T. Maas	Complete restructuring
2008-09-17	0.7	T. Maas	Fixed bookmarks. Added appendices. Restructured Problem Description. Restructured chapter 3, headings. Restructured chapter 4.
2008-10-16	0.8	T. Maas	Added introductions
2009-04-09	0.9	T. Maas	Final results added
2009-04-15	1.0	T. Maas	Final version. Added preface and abstract. Fixed references. Added appendices. Added variation tests.
2009-04-18	1.1	T.Maas	Minor explanatory additions to chapter 5. Minor changes in layout.

1 INTRODUCTION

Before we start describing the problem of automated schema matching for Universal Data Services for which the research is described in this document, we will give a description of the environment of Universal Data Services and we will introduce some other relevant terminology for this domain of research such as Semantic Web and ontologies. We start with a description of the company URBIDATA, for whom this research project was executed. In section 1.2 we will describe Universal Data Services. This is followed by a description of some relevant terminology and finally we will give a problem description.

1.1 URBIDATA

The application of modern data processing equipment in companies and governmental organizations keeps expanding and because of that, production processes are transforming into information processes. Therefore companies tend to use a lot of different information systems. Consequently both people and computer systems need to communicate to reach certain goals. A need for data-exchange and integration arises.

URBIDATA is a company that specializes in data integration with focuses on web services, operational data stores and data warehousing. Their primary focus is spatial data integration for government, cities and municipalities.
To realize this integration URBIDATA constructs information factories in cooperation with their clients.

An information factory [8] is a system that converts data into information units. The data is processed with various methods in order to enrich it:

- Data formats are converted
- Relevant data is selected
- Names, definitions and classifications are transformed
- Data from different sources is aggregated
- Data is integrated into a structured environment (a data model).

Government agencies use many different software packages. Each software package fills some administrative need for information and each software package probably has its own data source with its own data model. To create a more structured and centralized environment data warehousing and data integration are useful tools. URBIDATA offers a toolset to these agencies, so this can be achieved. The next section will explain something more about this toolset: Universal Data Services (UDS).

1.2 UDS

URBIDATA's primary goal is to help their clients with integration and metadata annotation of their data sources. To achieve their goal URBIDATA has developed a package called UDS for collecting data from various data sources and storing it in a more generic way using standardized target data models.

Many business processes have their own data structure and often these processes need to interact and exchange information. For each connection between business processes an interpreter is needed. The result is called a "Spaghetti" structure (Figure 1), which has a mangle of connections with all different interfaces. The goal of UDS is to create a centralized "Ravioli" system (Figure 2) where UDS provides all interfaces between the sources and enriches

the data with more information. This is achieved by adding metadata to the information already available. Metadata is data about data; for example the author, file size or creation date of a document. In Figure 2 the operational datastore filled by UDS resides in the center.

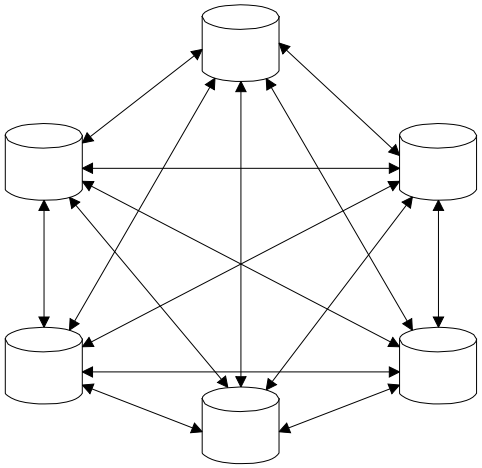

Figure 1: With n systems there are at least n(n -1) interfaces

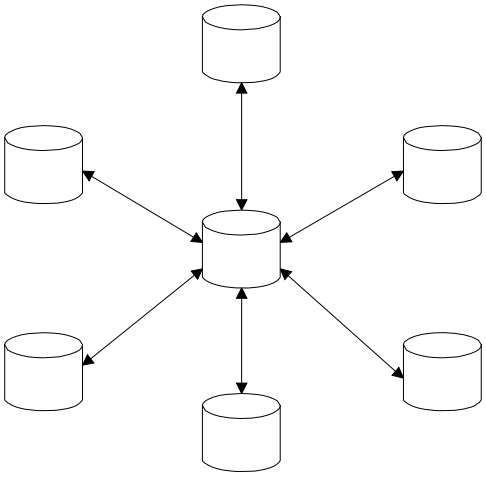

Figure 2: With n systems and UDS there are 2n interfaces

Obviously the terms “Spaghetti” and “Ravioli” refer to the number of connections within each model. The pasta spaghetti has many strings, you are unable to see where each string starts or ends; whereas in the pasta ravioli, you would be able to pick up a piece of ravioli very easily.

Reducing the number of connections improves the maintainability of the system, because there is less communication needed between components. A fully connected “Spaghetti” system has at least $n \cdot (n-1)$ interfaces; each component has at least one connection with all other components. A “Ravioli” system needs at most $2 \cdot n$ interfaces; each component is connected with the ODS (Operational Data Store) component in the center and the ODS component is connected with each component.

UDS currently provides 4 components: the Meta Data Manager (MDM), the Universal Data Integrator (UDI), Queue and the Broker. Each service has its unique function within UDS to ensure the “Ravioli” structure.

1.2.1 THE META DATA MANAGER

The purpose of the Meta Data Manager (MDM) is to manage data on the sources that are being used by UDS. It also provides a user interface which enables the user to enter more specific information about data sources and their objects and attributes.

A data model is a representation that describes the data structure of a source. MDM analyses the data structure of the data sources and enables the user to add more specific information to that data model in the form of metadata. For example the name of the author of a file can be added to the meta database, or the date when a source was added the integration process, or a short description of the source. Just about anything containing data can be a source, for example data repositories, XML files, AUTOCAD files, etc. The MDM can also

be used to execute search queries on the metadata, for example finding the origins of certain data sets within the integration process. Finally attributes can be selected from source models and transformations can be created and operations selected.

1.2.2 The Universal Data Integrator

With the UDI (Universal Data Integrator) a user can perform transformations on data using the metadata that was described using the MDM. In the UDI the transformation models are designed as follows:

- Transformations and operations can be executed on the selected attributes.
- Finally the data that was created can be integrated into the UDI data store.

The UDI works according to the STOI principle. See Section 1.3 for more details on STOI. The steps described above are captured in the form a process.

1.2.3 Queue

The Queue component acts as a scheduler. In the Queue component the processes defined in the UDI can be scheduled. The UDS Queue component handles the communication between the different UDS components and makes sure that they are synchronized, which prevents data from getting corrupted because multiple instances were operating on the same data.

1.2.4 The Broker

Last but not least the UDS Broker is the part of the system that acts as an Enterprise Service Bus (ESB) [16]. An ESB enables components to communicate with each other without dependencies, but by an event-driven and standards-based messaging engine. This makes UDS even more flexible; it uses this form of communication so different components can be “plugged into” the Broker.

Conclusively the Broker itself essentially is a kind of translation mechanism that translates incoming data into a format that is supported by UDS. For example when a new component is added to UDS, the Broker will interpret the output of this new component and translate it to a format that the already existing components in UDS can use.

1.3 Extract Transform Load

Data integration and data warehousing is generally based on the ETL principle [17]:

- Extracting data from sources
- Transforming the data to fit the business needs
- Loading the data into a data warehouse

UDS also uses this principle; with the MDM users can create transformation and integration processes that can become very complex, but are mostly ordered in an ETL fashion. URBIDATA defines this as STOI:

- Selection
- Transformation & Operation
- Integration

So ETL and STOI are actually the same. Selection is the process where a subset of data is selected from a data source for integration; Extract does this as well. Transformation & Operation uses metadata from a transformation model to transform the data to fit the target data model; Transform from ETL does this as well. Finally Integration writes the data to the target database; Load does this as well. The only difference that might be mentioned is that their semantic emphases are different, whereas Extract focuses on the actual extraction of the data, Select focuses on the selection process which data is going to be extracted from the source.

Next to classic ETL there is Spatial ETL, which does the same for spatial data. Spatial data is nothing more than geographic information. However this kind of data is more complex. It can contain geometric models, images and maps.

1.4 Environment

The MDM was originally created using Borland Delphi 7 and the Borland Database Engine (BDE). When Borland announced that they were planning to terminate the support for the BDE and a new geographical metadata standard (ISO19115) was introduced, the decision was taken to redesign MDM completely.

During this redesign process a number of problems needed to be solved. The Borland Database Engine was removed and a database independent structure was designed in order to enable MDM to interact with different kinds of database management systems. Also the metadata database structure was updated to meet the current standards; the CEN metadata standard needed to be replaced with the ISO19115 metadata standard.

1.5 Problem description

The thesis project was initially about adding transformations, operators and processes to the redesigned MDM. However there is also need for innovation.

Adding transformations and processes to an installation of UDS manually is a very time consuming process. Therefore research is necessary to discover whether some sort of automation is possible and also whether some form of machine learning might be useful in the automation process.

The initial study of the problem made clear that the focus of the project had to be smaller than initially anticipated. The research that would have to be done in order to automate the entire integration process would be too much time consuming to be able to finish within the time span of the project. After some deliberation with the project supervisors it was decided that automated schema matching would become the focus of the project. Of course this also implies that a way to store generated matchings needs to be researched, but this is not the main focus of this project.

Automated schema matching is a first step toward automated integration. In order to be able to identify transformation rules first the matching between two models needs to be known. In this thesis the research that was done on the subject of schema matching and a basic framework for automated schema matching for UDS are described. Also a prototype was developed in order to demonstrate the use of automated schema matching. A description of future development necessary to complete automation of integration processes can be found in section 6.2.

1.6 PROBLEM ANALYSIS

In order to automate an entire integration process, a number of problems needs to be solved. First of all, a data model consists of objects and attributes. These objects and attributes need to be matched against the target model that we have in mind. Fortunately URBIDATA uses standard target models, which has the advantage that the structure of the target model is well documented. But still this matching process can become very difficult since there is in principle no knowledge of the structure source model. The structure of the source model can differ a great deal from the structure of the target model. For example the source data model could have an object containing a person's name and another object containing the person's address, while the target data model contains an object representing a person which also contains the person's address. A human being can construct such matchings using natural insight, but when large data models are in play this consumes a lot of time. If this matching process is automated however, it has to be taken into account which types of reasoning a human being, often unconsciously, applies when creating a matching between two data models. What are those types of reasoning? Can they be automated, and if so, how can this be done? Can existing techniques such as semantic maps, machine learning, semantic web and ontologies be of any help here?

If a matching is found, it needs to be stored somehow. This turned out to be no trivial matter. As was mentioned before, matchings can be very complex in structure. In order to enable easy retrieval of matchings, thought has to be put into a structured way of storing matchings. Also, since it is never 100 percent sure that an automatically generated matching is correct, it is necessary to ask for user feedback on a generated matching. Therefore a user friendly way of presenting a matching is needed. An intuitive way of storing matchings would therefore be a great help in presenting the matching in an intuitive way to the user. Another question that arises is what should be done when a user does not approve a generated matching. Should a new matching be generated based on user feedback, or should we store not only the most likely matching but also some other, less likely but possible matchings, or should the user fix the matching?

When the correct matching is found, a transformation model needs to be created. The transformation model states the rules for conversion of the source data model to the target data model. These rules are called transformations. A transformation can be fairly simple; creating a copy. However a transformation can become complex very fast; for example attribute specific substring replacements, computations, interpreting and merging. Sometimes two attributes need to be merged into one attribute. Consider for example a postal code attribute containing the four numbers of a postal code and another attribute containing the two letters. In the target model these may need to be merged into one postal code attribute. Also type conversions may occur.

Consider for instance a currency that is represented as a float in the source model and that needs to be represented by a specific currency type (such as Euros) in the target models. These are just a few examples of the extensive list of possible transformations. Again we can note that a human being constructs these transformations based on natural insight, mostly based on knowledge that was obtained earlier. This suggests that some sort of knowledge base will be necessary in order to automate the process of creating a transformation model. What knowledge exactly does this knowledge base have to contain? What is the best way to represent this knowledge base? And what are the reasoning rules that need to be defined in order to be able to use this knowledge base to automatically create a transformation model? These are all questions that need to be handled when automating this part of the process.

1.7 OVERVIEW

The remainder of this document describes the research that was done in the area of automated data integration, the design, implementation details and testing of a component that automates the data integration in UDS and an evaluation of the solution that was offered in this document as well as an evaluation of the project. The research that was done in the area of automated data integration can be found in chapter 2. The scenario at Urbidata is described, weaving models are introduced as a way to represent the integration of models and the process of schema matching is explained. Chapter 3 give a sketch of the framework design. The usage of semantic web concepts, machine learning and topic maps is described, algorithms that can be used for structure matching are given and the use of weaving models, ontologies and machine learning in semantic matching is explained. Finally it is described how the results structure matching and semantic matching can be combined into one model. Chapter 4 describes implementation details such as data types, architecture, and dependencies. Chapter 4 also gives a description of how to set up, configure and use the component. An evaluation of the component, including a test scenario, is described in chapter 5. Chapter 6 concludes with the results of the project, a description of future work and an evaluation of the project.

2 AUTOMATED DATA INTEGRATION

This chapter describes research concerning automated data integration. In section 1 the scenario of this particular project is described. Section 2 describes weaving models as a way to represent integration between models. In section 3, the process of schema matching based on structure and semantics is described. This section also explains the use of user feedback, topic maps and value matching on metadata in this project.

2.1 SCENARIO

The setup of a data integration project at a client is a time consuming process. UDS allows the user to set up an operational data store manually. The remainder of this section will describe this scenario in detail. The next sections describe how this process could be automated.

2.1.1 UDS ARCHITECTURE

UDS is built using a client-server architecture using services in an Enterprise Service Bus environment. Figure 3 shows a model of the high level architecture of UDS. UDS can be divided into three blocks. The UDS Client, containing the user interfaces for the user, the UDS Server, containing the main components for communication with the databases, and the UDS Enterprise Service Bus, containing some components for communication between the services and client and server components.

In the Services block the Integrator engine is the most important component. It reads data from the data source and imports them to the UDS Oracle Spatial Database or exports data from the UDS Oracle Spatial Database to data targets. Moreover it transforms data when necessary.

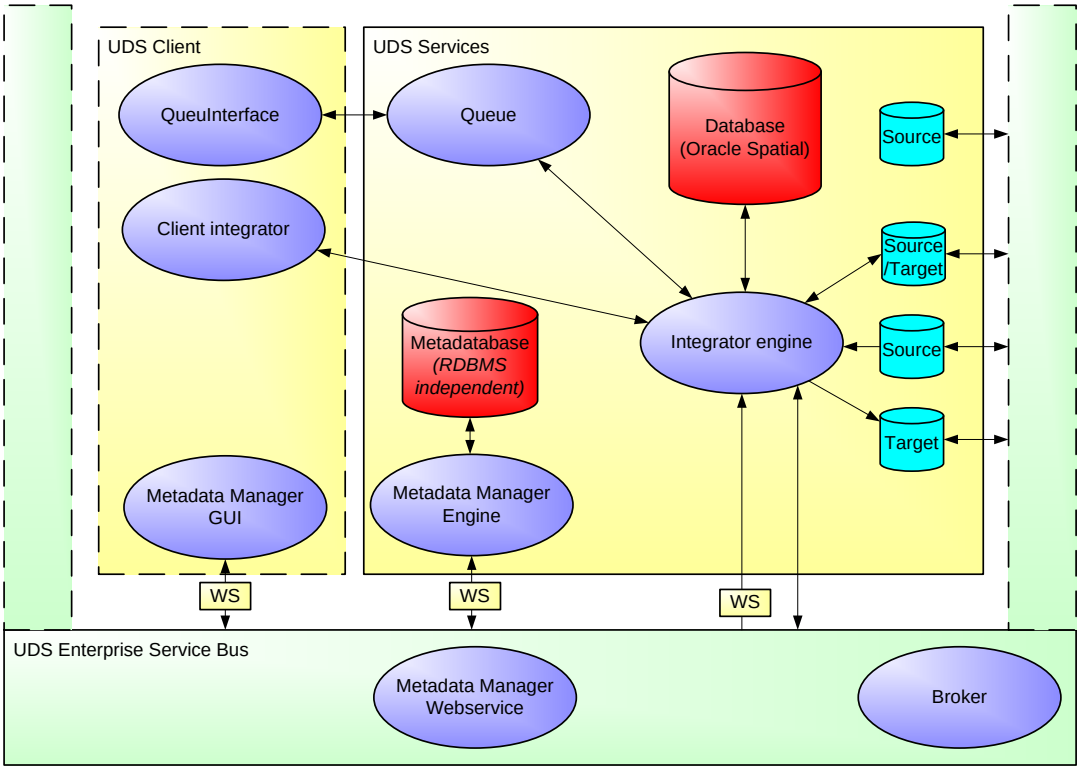

Figure 3: UDS High Level Architecture

The Integrator engine has a direct link with the Queue component. The Queue component schedules the tasks for the Integrator Engine. The reason for this is that for example the load of the system is lower when an integration process is started at night, when it is less likely to have a lot of database transactions going on.

Finally the Integrator engine communicates via the MDM Webservice with the MDM engine. The MDM engines task is to save the metadata that was collected over the data from the data sources to the Metadatabase. These descriptor fields can be filled out by the user, using the Metadata Manager GUI. In the Metadata Manager GUI the user is also able to define the transformation rules for the integration process.

The setting in Figure 3 is the most recent setup of UDS. The need existed to be able to put new MDM GUI's on top of the MDM engine; for example web-based GUI's. Therefore the webservice is a communication component handling the communication between the MDM engine and other components. In this scenario it is much easier to replace components; not just a GUI, but the Integrator engine as well.

The collection of interfaces like the MDM webservice and the UDS Broker is called the Enterprise Service Bus. An ESB uses an event driven and standard-based messaging engine to provide an abstraction layer for communication between different components.

2.1.2 "SELLING" UDS

A scenario where UDS is placed with a customer would look as follows. The customer would contact URBIDATA to set up UDS. In cooperation with an URBIDATA consultant each data source that the customer wants to integrate needs to be inspected thoroughly; for each source a schema is needed that tells the integrator how to interpret the data model of the source and what to do with it. Defining these schemata is a very time consuming process and can take up to several months.

URBIDATA has a set of standardized data models available, for example the NEN3610 [20] as source data model and RSGB [21], BAG (Basic registration for Addresses and Buildings) [22] and the ENC-S57 (GIS) standard [23] as target data model. These standards have been registered by Dutch or international authorities. Government agencies are strongly advised to use these standards. The goal of using the data standards is to structure data and information collection.

2.2 WEAVING MODELS

In order to automate schema matching a representation of matchings is needed. UDS already uses transformation models, which are triples of the following form:

$$(\{Attributes\},\{transformations\},\{Attributes\}).$$

We will introduce the notion of a weaving model, which enables us to use this existing transformation model and extend it in order to serve our goal of automating schema matching.
In [3] model weaving is defined as "a generic way to establish element correspondence". Weaving models can be used to translate source models into a target model. A weaving model is a schema that defines the relationship between two data models [3] (Figure 4). Such a schema defines exactly which source attributes corresponds to what target attribute and in what way. Because these schemata are considered as models, namely weaving models, they can be enriched with metadata. Therefore the mappings between source and target models gain in expressiveness. Furthermore since mappings are considered as models the mappings also become more generic and gain in flexibility. They can easily be updated, data can be altered, metadata can be altered, added or removed, etc. A transformation model for instance does not have this flexibility because its structure is predetermined; it consists solely of a set of source attributes, a set of target attributes and the transformations, no extra information can be added.

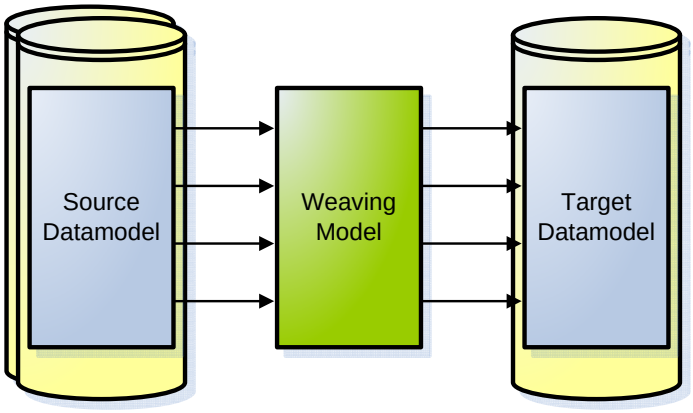

Figure 4: a standard weaving model situation

UDS uses transformation models. A weaving model can be used by the MDM to suggest an initial transformation model. Since there is no accepted formal definition of a weaving model [3], we will define the structure of a weaving model as is most convenient for this project.

We assume a data model to be a directed graph $G = (V, A)$. The set of vertices V denotes model elements. A model element from V has an identifier and a value. The identifiers are URIs and the element value is allowed to be of any data type. The set of directed edges A denotes associations between model elements.

As stated before a weaving model connects two data models using a mapping. Let $DM_1 = (V, A)$ and $DM_2 = (V', A')$ be the source and target data model respectively. Given elements $e_1 \in V$ and $e_2 \in V'$, the connection (e_1, e_2) is denoted by the triple (e_1, Wm, e_2), where Wm is a directed graph; $Wm = (Vw, Aw)$. A complete weaving model consists of a set of these triples; the resulting graph is not connected. Figure 5 depicts such a directed graph; attributes and structure are explained in more detail in section 3.7.1.

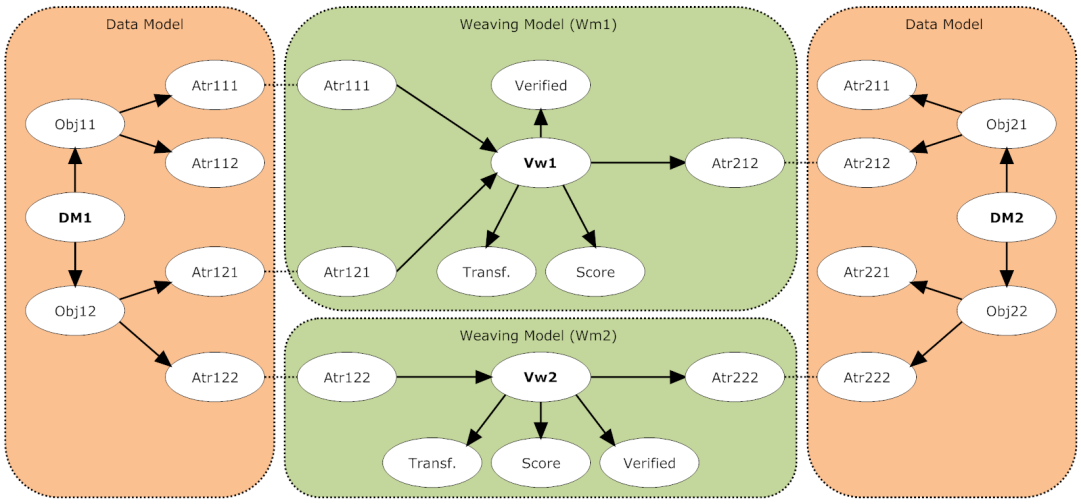

Figure 5: A weaving model directed graph

The following examples and all further examples used in this thesis contain Dutch words, because real strings from existing data models are used.

For example two source attributes POSTK_A and POSTK_N represent a postal code and the target attribute is PC. Then the weaving model could look something like

```
PC ← CONCAT(POSTK_A, POSTK_N)
```

This expression specifies that the value of the target attribute PC is constructed from the concatenation of the values of the two source attributes POSTK_A and POSTK_N.

Below in Figure 6 the example is depicted as a directed graph, where $V = \{POSTK\_A, POSTK\_N\}$ and $V' = \{PC\}$.
Then with the elements $e_1 \in V$ and $e_2 \in V'$ two triples can be created $(POSTK\_A, Vw_1, PC)$ and $(POSTK\_N, Vw_1, PC)$.

Where $Wm_1 = (Vw, Aw)$ with
$Vw = \{Vw_1, Transf(CONCAT), Score, Verified\}$
$Aw = \{\langle Vw_1, Transf(CONCAT)\rangle, \langle Vw_1, Verified\rangle, \langle Vw_1, Score\rangle\}$

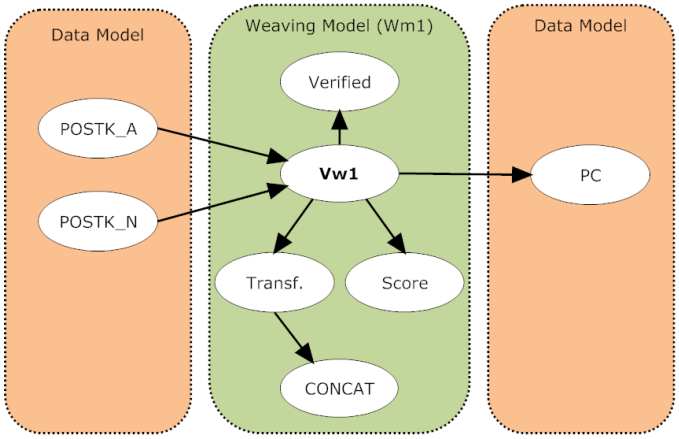

Figure 6: A weaving model for PC ← CONCAT(POSTK_A, POSTK_N)

2.2.1 Attribute to Attribute

One of the simplest and most common forms of data model conversion is to copy the value of one attribute to another.
For example STRAATKODE to STR_CODE

```
STR_CODE ← STRAATKODE
```

2.2.2 TRANSFORMATIONS

A transformation is the situation where some action is taken on one or more source attributes to alter their values in a certain way, before the values are copied to the target attribute.
An example of this would be the concatenation example from above or for example the following transformation on a single source attribute

```
HOEK_HUISNR ← REPLACE(“,”; ”.”; VRY_VELD2)
```

This carries out a substring replacement on the values of VRY_VELD2 to produce the value for HOEK_HUISNR.

2.2.3 OPERATIONS

An operation is actually exactly the same as a transformation. However UDS still distinguishes these two. A transformation can be carried out during each step of a data model integration process and an operation is carried out in bulk after the integration process. Bulk means all data has to be read before the operation can be executed instead of one data element at a time.

An example of a weaving model operation is the generation of unique MD5 identifiers. MD5 is a cryptographic hash function. A MD5 hash is usually expressed as a 32-character string of hexadecimal numbers. Assuming all data is unique, then hashes are unique, making it easy to use them as identifiers.

```
MD5_ID ← MD5(self)
```

2.2.4 AUTOMATED WEAVING MODEL GENERATION

To reduce the time on customer projects the generation of the weaving models could be automated instead of creating weaving models manually. For this to work introduce a few assumptions.

First the target data models that URBIDATA uses are mostly standardized data models. These models are tweaked when the customer specifically asks for it. Secondly it is assumed that the source data models are completely random and do not contain any metadata whatsoever. Therefore there is no need to incorporate the use of stored metadata on the target data models and the focus will be on the structure of the source data models.

A weaving model is a solution to the problem of how to store a schema matching, as was noted in section 1.6. The decision was made to store not just the most likely matching, but also some less likely matchings. Since in a weaving model the score of each connection is stored, it is possible to retrieve generated weaving models in order of most likely matching to less likely matching. Also we noted the problem of necessary user feedback on generated matchings. A solution to this problem is presented in section 2.3.2.1.

This thesis will focus on two branches of weaving model generation; schema matching based on structure and schema matching based on semantics. These will be discussed in the following section.

2.3 SCHEMA MATCHING

A schema (in this case a data model) consists of objects and objects consist of attributes. This is the general structure of a data model. For example a simple relational database has tables (objects) and each table consists of fields (attributes).
Now suppose we have two arbitrary data models A and B, of which we want to map A to B. In the beginning we do not know which object from A needs to be mapped to which object from B (Figure 7). So it might be wise to work with elimination to find out where each attribute should be mapped to.
In the elimination method initially all objects are assumed to be potentially equal and during this process the mismatches are removed one at a time.

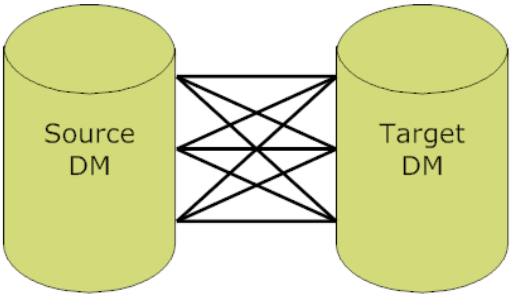

Figure 7: Mapping two arbitrary data models on object level

However, to know whether two objects match, the schema needs to be evaluated on an attribute level (Figure 8). As a result of connecting all objects, all attributes are initially connected as well.

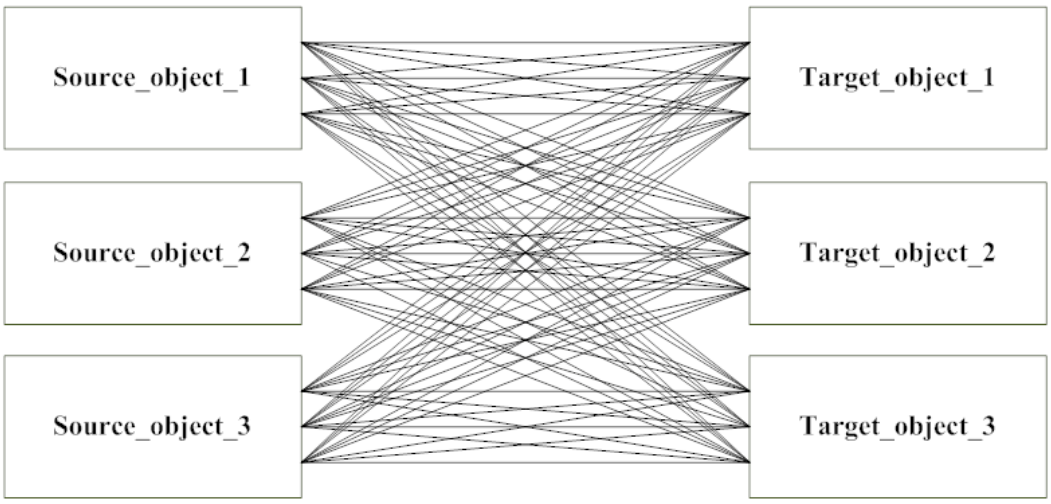

Figure 8: Mapping two arbitrary data models on attribute level

Now a score is given to each connection. The mapping, a set of connections, with the highest score would be the most probable mapping, the mapping with the one to highest score a bit less probable and so on.

Now a method is needed to give a score to each connection. As was mentioned in section 1.6, human beings can create matchings based on natural insight. How do they actually do this? One way of comparing source and target attributes to each other is on the basis of their structure. This results in structure matching, which is described in section 2.3.1. Human beings are also able to compare the meaning of a source attribute to that of a target attribute. This results in semantic matching, which is described in section 2.3.2.

2.3.1 STRUCTURE MATCHING

In this section several methods are described that can be used in order to match a source data model to a target data model based on their structure. These methods of matching are used to score connections between attributes.

A way to score a connection between two attributes is string comparison. The name of each object and the fieldname of each attribute could be compared to any other name of a target object or fieldname of a target attribute.

Another way to score a connection is by means of type matching. When two attributes have the same or a similar type, some value is added to the score. For example when the type of the source attribute is a string of a constant length and the target is a string of variable length, we can assume that their types match.

These scores are used to create scores on an objectlevel. Therefore creating mappings which contain semantics. Following that scoremodels can be created from these mappings. To add more value to the score of the object connections the highest total score of the attribute connections from one object to another is evaluated. To get the highest total score a search is conducted for the optimal mapping of attributes between two objects that has the maximum sum of the score of those attributes.

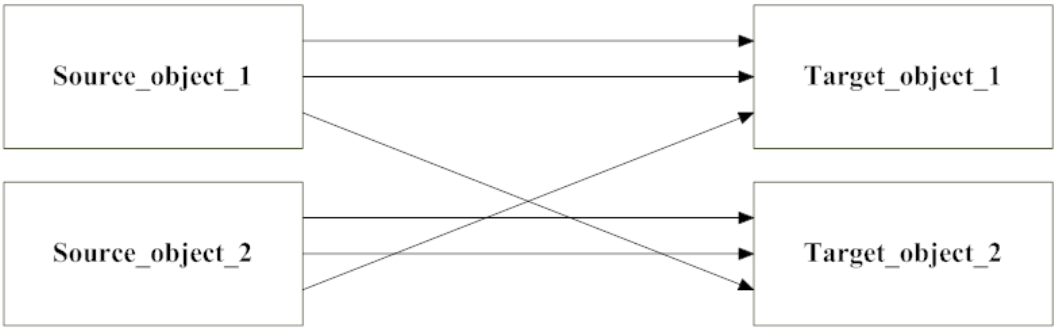

Figure 9: Comparing the maximum score of attribute connections results in a mapping

Schema matching results in a simple mapping. It suggests WHERE data should be integrated, not HOW it is to be integrated. Therefore a Semantic History is used that can achieve a HOW.

2.3.2 SEMANTIC MATCHING

In the past years URBIDATA has set up operational datastores of mid offices in many UDS projects. Each of these projects has a large amount of transformation rules for integrating data sources. This information can be very useful in determining a weaving model and which source attributes relate to which target attributes and how they relate.

Each source data model in combination with a set of transformation rules and a target data model gives us a lot of information how future data models can be integrated. A weight property is used for each source attribute, transformation rule, target attribute combination to determine the most common used combination and match these combinations against new scenarios. We will call this method *History Matching*.

In order to design such a semantic matching component, storage and retrieval of previously generated weaving models needs to be taken into account. An ontology is a nice way of dealing with storage and retrieval of semantically annotated objects such as weaving models. In order to assign appropriate weights to history objects machine learning techniques are used to update weights after each successfully completed integration project. Section 3.4 will elaborate on the usage of machine learning techniques and ontologies in history matching.

The next section describes two important methods used in History Matching, user feedback and value matching. Furthermore we will describe a standard for storing data about data models, which is useful during History Matching.

2.3.2.1 USER FEEDBACK

To make the history matching method work, a database is needed containing relevant data and weaving models of previous projects. Relevant means that the evaluation of the information domain with new projects should result into a weaving model as complete as possible.

However new projects might contain data model objects that haven't already been added to the history. The idea is to offer the user a provisional match. Such a match is given to the user with a certainty between 0 and 1. The user can now easily see which transformation matches need evaluation. When an evaluation has been approved by a user, the history is updated using machine learning techniques so the information can be used in future matches.

This user feedback can also serve as a method for general approval or dismissal of the matchings in a weaving model. As was stated in section 1.6 user feedback is necessary since we can not guarantee 100% correctness of the automatically generated weaving model. User feedback is now used for approval of automatically generated weaving models as well as for supervising the learning process in the history component.

2.3.2.2 VALUE MATCHING

In value matching, metadata is added to the history items that define a regular expression which describes the values of previous data models. This way we can extend the semantic database that is used in history matching.

An object contains attributes with values. The result of a group of attribute values is a set of strings. All values of all types of attributes are converted to strings. This makes it easier to compare the values in general. The set of strings can be converted to a regular expression by tokenizing the characters. For example if we look at the result of the group for the attribute postcode, we get a set of strings (5993AX, 5612LC, 5616GB, 5612AZ). If we tokenize this set we get 5{4}, 9{1}/6{3}, 9{1}/1{3}, 3{1}/2{2}/6{1}, A{2}/L{1}/G{1}, X{1}/C{1}/B{1}/Z{1}, the number between the accolades indicate the number of times the character was encountered on that position.
Therefore the regular expression probably would be [0-9]{4}[A-Z]{2}, which means 4 numbers followed by 2 capital letters. Such a regular expression can be stored in metadata of an attribute and matched against future values of that attribute.

3 FRAMEWORK DESIGN

Chapter 2 describes several methods and algorithms to construct a weaving model from source and target data models. These methods and algorithms are part of the framework Weaving Model Generator (WMgen). The purpose of this framework is to generate a weaving model that contains certainties as metadata, so the user can evaluate the model. This chapter provides a detailed design of the WMgen.

Section 1 describes the use of semantic web concepts in this design. In section 2, a conceptual, high level design is given. Section 3 gives a detailed description of structure matching in WMgen and sections 4 through 7 describe semantic matching in WMgen in detail. Section 8 concludes with a description of how structure matching and semantic matching are combined to create one model.

3.1 SEMANTIC WEB

The worldwide web is designed to be used by humans, not by computers. Semantic Web enables computers to use information that is stored on the web without help from a user. As its name states, it enables computers to have a sense of the semantics of the data stored on the web. Using the technology of Semantic Web, meaning can be added to the content of a web page, making it possible for machines to process knowledge using deductive reasoning and inference. This way, Semantic Web facilitates automated information gathering, which will prove useful in this project.

Semantic web uses technologies such as Extensible Markup Language (XML), Resource Description Framework (RDF) and Web Ontology Language (OWL). XML can be used to tag information; a user can define his own tag such as “zip code” or “street name”, enabling a program or script to interpret the data as a zip code or a street name. But since a user can define his own tag, the writer of the program or script that interprets the XML written by the user needs to know which tags the user uses and what he uses them for. Here, RDF and ontologies come into play. RDF expresses meaning encoded in sets of triples. Within these triples it is expressed that objects have properties with a specific value. For instance the object John has the property that he lives somewhere and the value of where he lives is Eindhoven. The triple (“John”, “lives in”, “Eindhoven”) then expresses the meaning that John lives in Eindhoven. Using RDF, essentially means that we are now able to express the meaning of a relationship between two objects defined by tags in XML. But there is still the problem that we do not know what the meaning of those objects (defined by the tags) is. Semantics need to be defined for the keywords that are used as tags in XML. This is done in an ontology. [7]

3.2 CONCEPT

The final product of this project will be a prototype, a stand-alone piece of software capable of generating a weaving model correctly for a certain set of data models. It will be implemented as a Java web service. The in- and outputs will be in XML format. Java has been chosen for its platform independence and because it is the programming language that is becoming more and more important in future products of URBIDATA. Several libraries for database communication, algorithms and data model conversion techniques will be used. More on implementation issues in chapter 4.

The Weaving Model Generator takes two data models as input. The output is a weaving model which is presented to the user. The user then evaluates the model. The corrected weaving model can then be used as input to the WMgen to update the history.

The history information is basically stored in an ontology. Therefore a RDF knowledge base is needed for storing, inferencing and querying ontologies. In short, it is a database for ontologies.

Figure 10 gives a graphical representation of the WMgen framework and how it interacts with users. The WMgen takes two data model representations as input and outputs a weaving model representation. This weaving model is presented to the user via a user interface that clearly shows the certainties of the connections. The user then evaluates the presented model and the acceptation information is added to the history, which is used by the WMgen.

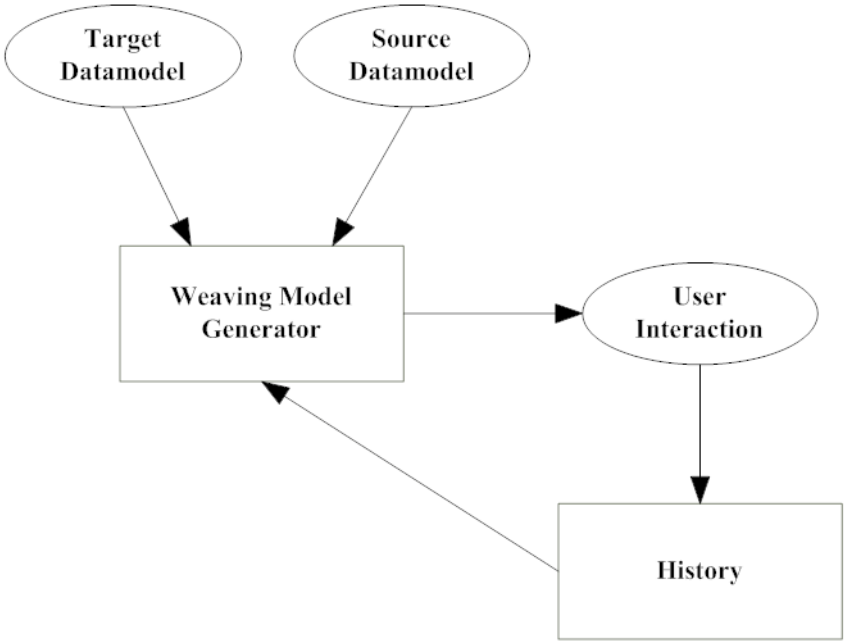

Figure 10: WMgen framework user interaction

In the thesis project the components of the WMgen can be split up into syntactic scorers, which are scorers using string and structure matching algorithms, and semantic scorers, which contain scorers that use semantics to reason about models, having for example history matching and value matching algorithms.

Figure 11 is a graphical representation that shows these 2 types of components for the WMgen. The components operate individually, so new components and match methods can easily be added to the WMgen.

Then there is the final component, the weaving model comparer. This component takes the results of the subcomponents and compares the scores of the resulting weaving models to determine the weaving model with the highest overall score.

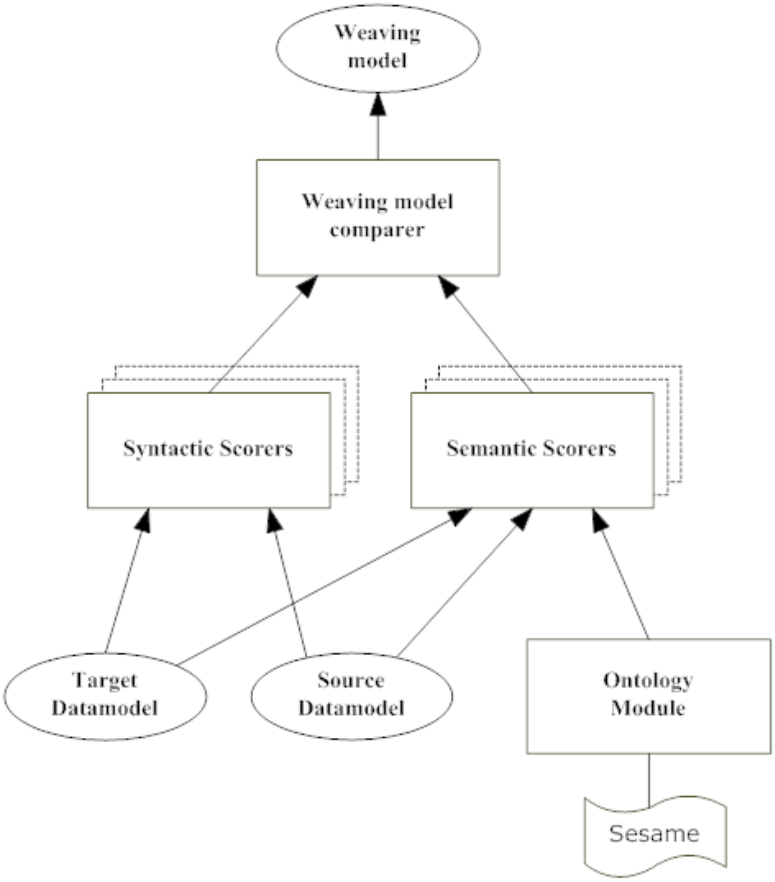

Figure 11: High level model of the WMgen scorers and comparer

In the following sections the Syntactic and the Semantic Scorer types are described. Finally the Weaving model comparer is described.

3.3 SYNTACTIC SCORER TYPES

This section will explain more about the schema matching done by syntactic scorers of the type. Syntactic scorers use the structure of the model to construct weving models. These components takes a source attribute or object name and a target attribute or object name as input and give a score as output. The two attributes can differ in two ways. For example consider the following source and target objects

```
ADR5_WOONPLAATS          SA_WOONPLAATS
ADR5_STRAAT              SA_OPENBARERUIMTE
ADR5_ADRESCYCLUS         SA_NUMMERAANDUIDING
```

As can be seen from the examples the object names might be completely different words and another possibility is that they contain prefixes, underscores and abbreviations. To compare strings, that differ in such a way that a computer could not tell that they are actually the same, there is a need for some kind of translation mechanism. A thesaurus is some kind of dictionary that stores synonyms and homonyms of words. The thesaurus can be used to generate more matches with the target attribute to enrich score (Figure 12). Only the creation of synonyms for the source attribute is required, because the target attribute resides in a fixed standardized model.

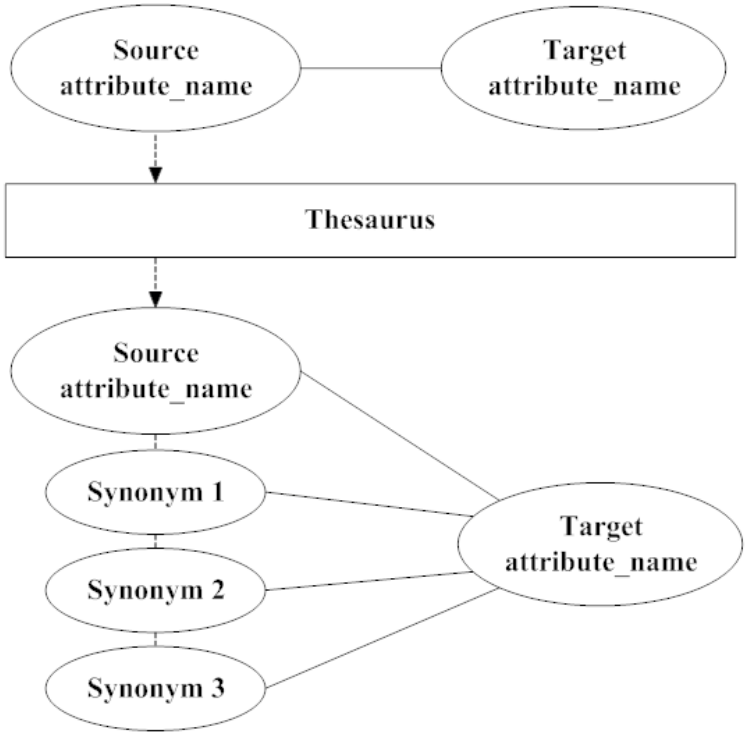

Figure 12: Creating a better score by using a thesaurus

Also it can be assumed that object and attribute names might contain prefixes. It is less likely to find synonyms for strings with (random) prefixes. Therefore these prefixes should be removed. To find a prefix all object names or all attribute names from an object are compared. If the first *n* characters of each string is the same, this is called a prefix.

The Syntactic Scorer's task is to give an estimate score of a match between two attributes or two objects. To achieve this first a thesaurus as described above is used to create better matches. After all synonyms have been generated, the source to target connections are passed through a series of score algorithms that independently add score to the matching process.

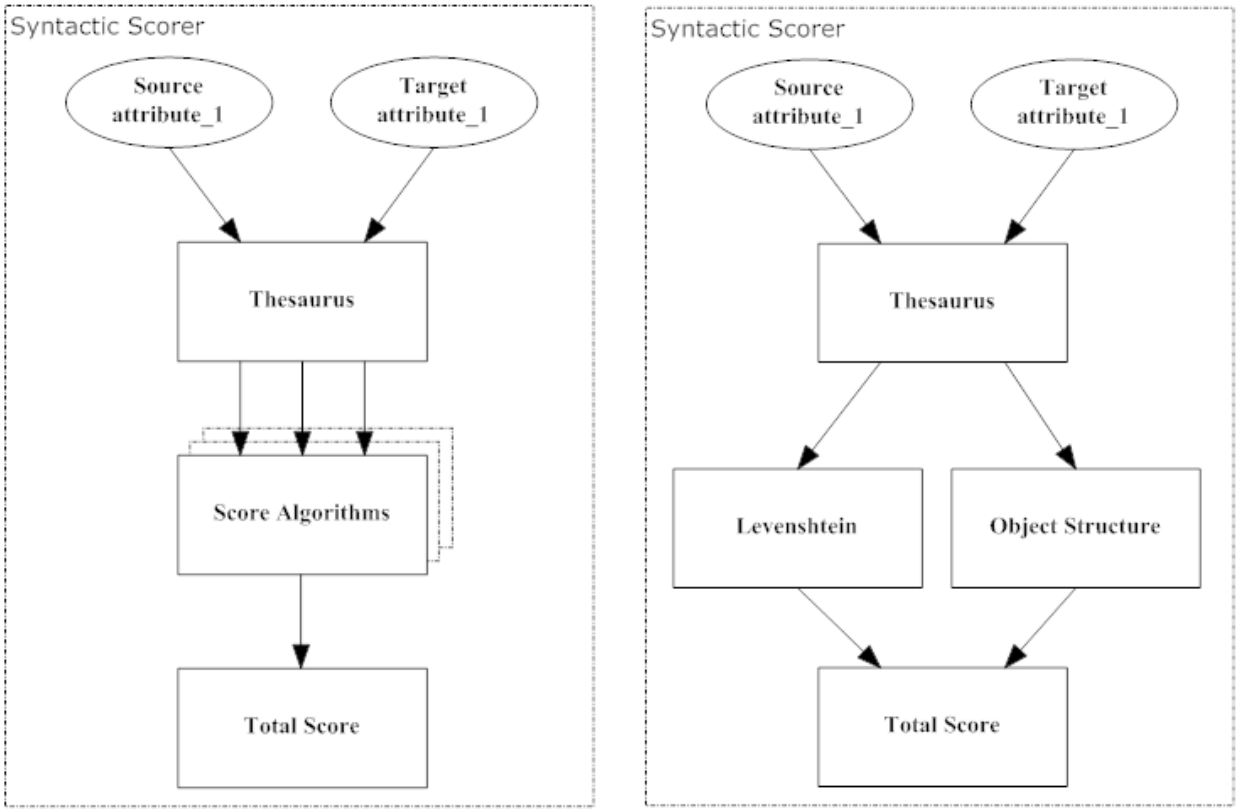

Figure 13: Syntactic Scoring using schema comparison algorithms

Figure 13 depicts the behavior of the Syntactic Scorer with two examples of scoring algorithms that were used for this project. For future purposes more algorithms might be added to get a more accurate score.

Algorithm $StructureScorer(source\_attribute, target\_attribute)$

1. ▷ Returns the overall structure score of the connection between $source\_attribute$ and $target\_attribute$.
2. $source\_synonyms \leftarrow CreateSynonyms(source\_attribute)$
3. $score_1 \leftarrow \emptyset$
4. **for** each $synonym \in source\_synonyms$
5. **do**
6. $score_1 \leftarrow score_1 \cup \{Algorithm_1(synonym, target\_attribute)\}$
7. $score_2 \leftarrow \emptyset$
8. **for** each $synonym \in source\_synonyms$
9. **do**
10. $score_2 \leftarrow score_2 \cup \{Algorithm_2(synonym, target\_attribute)\}$
11. $score_3 \leftarrow \emptyset$
12. **for** each $synonym \in source\_synonyms$
13. **do**
14. $score_3 \leftarrow score_3 \cup \{Algorithm_3(synonym, target\_attribute)\}$
15.
16. ...
17.
18. $score \leftarrow \{\max(score_1), \max(score_2), \max(score_3), \ldots\}$
19. $total\_score \leftarrow avg(score)$
20. $transformations \leftarrow \emptyset$
21. $verified \leftarrow false$
22. $Vw \leftarrow \{wm, total\_score, transformations, verified\}$
23. $Ew \leftarrow \{(wm, total\_score), (wm, transformations), (wm, verified)\}$
24. $Aw \leftarrow \{(source\_attribute, wm), (wm, target\_attribute)\}$
25. $WeavingModel \leftarrow (Vw, Ew, Aw)$
26. **return** $(source\_attribute, WeavingModel, target\_attribute)$

```
Algorithm CreateSynonyms(name)
1.  ▷ Returns a set of known synonyms of name
2.  synonymset ←{name}
3.  synonymset ←synonymset ∪ {RemovePrefixes(name)} synonymset ←synonymset ∪
    {TranslateAbbreviations(name)} synonymset ←synonymset∪{RemoveUnderscores(name)}
4.  thesaurusset ←∅
5.  for each element ∈ synonymset
6.      do
7.          { let Thesaurus(element) denote the set of synonyms that the thesaurus returns
            for element }
8.          thesaurusset ←thesaurusset ∪ Thesaurus(element)
9.  synonymset ←synonymset ∪ thesaurusset
10. return synonymset
```

Figure 14: A pseudo code algorithm for the Syntactic Scorer Types

Figure 14 gives a more detailed description of what happens in Syntactic Scorers. Syntactic Scorers create a score listing for each possible weaving model rather than the weaving model with the highest score, because more scorer modules can be used to generate other scored matchings. In a later stadium these score listings can be compared to determine the best weaving model.

3.3.1 MATCHING ALGORITHMS

The main function of the Syntactic Scorers is to score all the connections between the source and target data model on a structure level. Basic metadata information like field names, field types and object and attribute relations, is used to create matchings. This is done using the algorithms described below.

3.3.1.1 JARO & JARO-WINKLER

First we present the Jaro distance which is a method used in record linkage especially in order to check for spelling deviations. The Jaro-Winkler distance [1] is a variant on the Jaro distance. Given two strings, the Jaro-Winkler distance gives a result between 0 and 1, the higher the result the more similar the two strings are.

The classic Jaro distance d_j is calculated as follows:

$$d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$$

Where

- s_1 and s_2 are the two strings.
- m is the number of matching characters between s_1 and s_2 with a limit on characters that have positions not farther apart than

$$\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1$$

- t is the number of transpositions, which is the number of matching characters on different positions divided by two.

The Jaro-Winkler distance uses a prefix scale which gives more favorable ratings to strings that match from the beginning for a set prefix length *l*. So the Jaro-Winkler distance d_w is:

$$d_w = d_j + \left(l \cdot p \cdot \left(1 - d_j\right)\right)$$

Where

- d_j is the Jaro distance of the two strings.
- *l* is the length of the common prefix with a maximum of length 4.
- *p* is a constant scaling factor. The standard value for this constant is 0.1.

With example strings s_1 STRAATKODE and s_2 STR_CODE, we find

- $m = 6$ *(STR, ODE)* within the match window of 4
- $|s_1| = 10$
- $|s_2| = 8$
- $t = 0$
- Then $d_j = \frac{1}{3}\left(\frac{6}{10} + \frac{6}{8} + \frac{6-0}{6}\right) = \frac{47}{60} = 0.7833$

Thus the Jaro-Winkler distance, considering a prefix length $l = 3$ *(STR)*, would be $d_w = 0.7833 + \left(3 \cdot 0.1 \cdot \left(1 - 0.7833\right)\right) = 0.8483$

	S	**T**	**R**	**A**	**A**	**T**	**K**	**O**	**D**	**E**
S	1	0	0	0	0	0	0	0	0	0
T	0	1	0	0	0	1	0	0	0	0
R	0	0	1	0	0	0	0	0	0	0
_	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0
O	0	0	0	0	0	0	0	1	0	0
D	0	0	0	0	0	0	0	0	1	0
E	0	0	0	0	0	0	0	0	0	1

Figure 15: The Jaro-Winkler matching window of size 4 for the example

3.3.1.2 LEVENSHTEIN

One method that could be used is approximate string matching. There are several approximate string matching algorithms. The Levenshtein distance [2] algorithm is the most popular algorithm. For the WMgen we will use the Levenshtein algorithm. Vladimir Levenshtein described this distance in 1965. It represents how similar two strings are by the minimum number of operations needed to transform one string into another. The operations Levenshtein contains are insertion, deletion and substitution.

For example with s_1 = STRAATKODE and s_2 = STR_CODE, the Levenshtein distance would be 4, because STRAATKODE can be transformed to STR_CODE by two substitutions *(SUB(A, _); SUB(A, C))* and two deletions *(DEL(T); DEL(K))*. Levenshtein calculates a (m+1, n+1) matrix where the bottom right value represents the distance. The matrix for this value is depicted in Figure 16.

Basically this means that the algorithm walks through a distance matrix *d* by iteration. When we initiate $d[0,0]=0$, the value in each position can be calculated as follows:

$$c[i,j]=if\ \ (s_1[i]=s_2[j])\rightarrow 0\ \ else\ \ 1$$

$$d[i,0]=i$$
$$d[0,j]=j$$
$$i,j>0: d[i,j]=\min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+c[i,j])$$

		S	T	R	_	C	O	D	E
	0	1	2	3	4	5	6	7	8
S	1	*0*	1	2	3	4	5	6	7
T	2	1	*0*	1	2	3	4	5	6
R	3	2	1	*0*	1	2	3	4	5
A	4	3	2	1	*1*	2	3	4	5
A	5	4	3	2	2	*2*	3	4	5
T	6	5	4	3	3	*3*	3	4	5
K	7	6	5	4	3	*4*	4	4	5
O	8	7	6	5	4	4	*4*	5	5
D	9	8	7	6	5	5	5	*4*	5
E	10	9	8	7	6	6	6	5	*4*

Figure 16: Levenshtein distance STRAATKODE to STR_CODE

So the Levenshtein distance in this example equals 4. To determine a score within the domain [0,1], the distance is divided by the maximum length of the two strings. Thus

$$score[s_1,s_2]=1-\frac{levenshtein[s_1,s_2]}{\max(strlen(s_1),strlen(s_2))}=1-\frac{4}{10}=0.6$$

3.3.1.3 SOUNDEX

Other methods of string matching are phonetic algorithms, which compare strings by the way they sound. SoundEx [14], developed by Robert Russell and Margaret Odell, is such an algorithm. It is based on the pronunciation of the English language. SoundEx tries to code strings in such a way that they have the same representation when they sound similar. Here an adaptation [13] of

SoundEx for the Dutch language is presented, because most of the data models used by URBIDATA use the Dutch language.

The English algorithm takes the following steps to encode a string:

- The first letter is kept.
- Letters that sound the same are replaced by numbers using the following table:

b, f, p, v	1
c, g, j, k, q, s, x, z	2
d, t	3
l	4
m, n	5
r	6

- All characters that are not in the table are removed.
- All double adjacent numbers are removed from the resulting string.
- The resulting string is limited to a maximum of 4 characters.
- Zero's are added to the resulting string until the string equals 4 characters.

The Dutch variant is similar; a replacement step is added and the lookup table is altered; now the steps to encode a string are denoted as:

- The first letter is kept.
- Carry out the following string replacements as a conversion between English and Dutch sounds:
-

QU	KW
SCH	SEE
KS,KX	XX
KC,CK	KK
DT,TD	TT
CH	GG
SZ	SS
IJ	YY

- Letters that sound the same are replaced by numbers using the following table:

b, p	1
c, g, s, k, z, q	2
d, t	3
f,v,w	4
l	5
m, n	6
r	7
x	8

- All characters that are not in the table are removed.
- All double adjacent numbers are removed from the resulting string.
- The resulting string is limited to a maximum of 4 characters.
- Zero's are added to the resulting string until the string equals 4 characters.

So again with the example STRAATKODE and STR_CODE, the SoundEx codes are S373(23) and S372(3) respectively.

As can be seen the results of SoundEx aren't easily comparable to other string matching algorithms. Therefore Jaro-Winkler is used as presented in section 3.3.1.1 on the SoundEx codes for the source and target attribute names to determine a score within the [0,1] domain.
In the example the score for SoundEx using JaroWinkler to compare the two codes will yield 0.9381.

3.3.1.4 TYPE MATCHING

Type matching is a simple mapping algorithm. The matching of two types has a score value that represents the result of the match. The result is at most 1 when two types are equal.

The other scores are deducted from a data type hierarchy. The data types as currently used in UDS are used as a basis. Figure 17 depicts a hierarchy where a type is a subtype when connected to another type, where *binary object* is the highest type and *string (fixed)* and *string (var)* are equal, but different.

ID	NAAM
1	Integer
2	Floating point
6	Spatial
7	Boolean
8	Datum
10	Currency
11	String
12	Virtual String
13	Spatial Point
14	Spatial Polyline
15	Spatial Polygon
16	SQL Timestamp
17	Spatial Annotation
18	Spatial Symbol
19	Spatial Cover

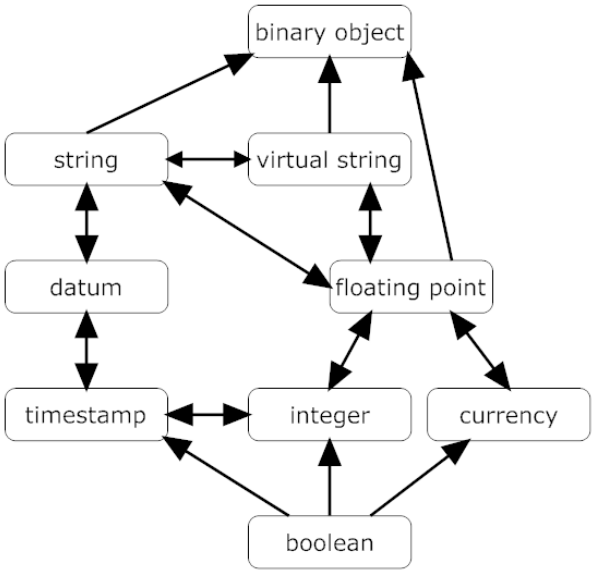

Figure 17: Oracle data type hierarchy

We use Figure 17 to construct a table; the score of a comparison between two types equals $½^{distance}$. So for example the score of a boolean/float comparison equals $½^2 = ¼$, and the score from string to binary object equals ½. However the score from binary object to string equals 0, because this conversion is difficult therefore there is a one way directed arrow.

3.3.1.5 OBJECT STRUCTURE

The Object Structure algorithm differs from the other algorithms used in the Match Scorer, because it uses the current attribute connection scores to calculate a score for object connections.

It takes the average score *a* of the set of attribute connection scores *c* between two objects o_1, o_2 and multiplies it by the number of attribute connections that

have a score above a predetermined threshold *t* divided by the total number of attributes that o_1 and o_2 share. The threshold can be adjusted to get better results from object structure analysis.

For the Object Structure score we now find
$a = StringMatch(o_1, o_2)$

$$object\_structure\_score = a \cdot \frac{\langle \#s_1 : s_1 \in c : s_1 > t \rangle}{\langle \#s_2 : s_2 \in o1.attributes \rangle + \langle \#s_2 : s_2 \in o2.attributes \rangle}$$

So to compare two objects using the object structure algorithm we use for example the score model depicted in Figure 18. Then if the threshold $t = 0.5$, Suppose $a = StringMatch(S\_o\_1, T\_o\_2) = 0.71$, then

$$object\_structure\_score = 0.71 \cdot \frac{2}{3} = 0.4733.$$

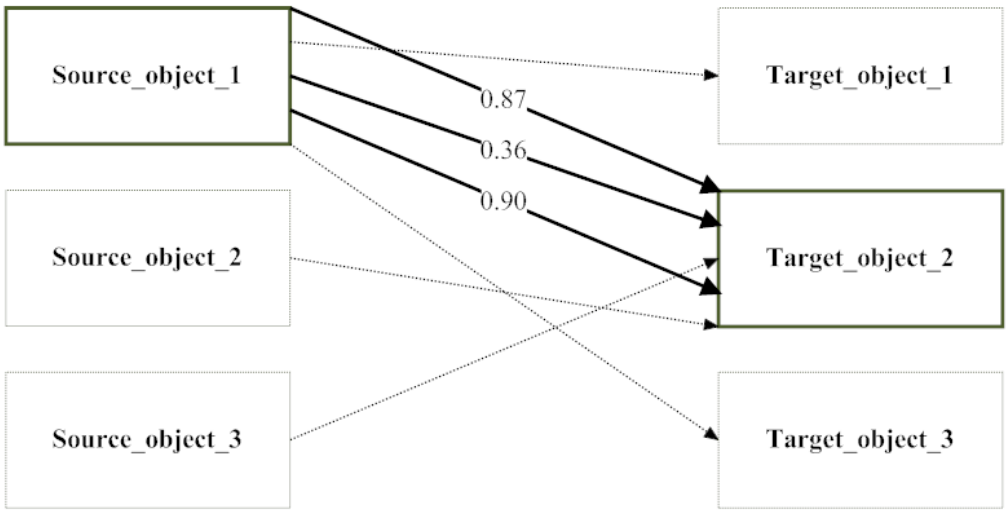

Figure 18: Object structure example

3.4 ONTOLOGIES

In philosophy the term ontology is used as a theory that defines the nature of existence, a theory that states what types of things exist. In Semantic Web, ontologies are used to define what things are; it defines to which class of objects an object belongs and what type of relations it can have with other objects.

Using the example of John who lives in Eindhoven, we can specify that “John” is a person, “Eindhoven” is a city and that persons can have the “lives in” relation with a city. Defining relations between objects and restraints on what types of relations are possible between objects (for instance a city object can not have a “lives in” relation with a person object) makes it possible for a machine to interpret and manipulate the information that it has access to. [7]

There is no need for the writer of a program or script to know what the custom tags of an author of a document represent, because the program itself is able to access the meaning of the document’s content through RDF and ontologies. The writer’s only concern is what information he would like his program to find and manipulate, not how someone else has encoded this information.

3.5 Machine Learning

As the term states, machine learning is a technique that is used in order to give machines the capability to learn. Learning here includes the learning of a wide variety of tasks in a large number of domains. It could be defined in general as the learning of an ability to perform new tasks or to perform old tasks better than before. This learning process is guided by changes produced during the learning process and by the learning process [9].

There are several paradigms that can be identified in machine learning. The four major paradigms are inductive learning, analytic learning, genetic algorithms and connectionist learning methods.

The inductive learning technique is based on induction of a general concept from existing instances of the concept and counterexamples which are not instances of the concept. Basically inductive learning is learning from examples.

Analytic learning in contrast uses few examples to learn from. It uses the underlying domain theory of the problem and deductive reasoning based on past experience in problem solving to guide the solving of a new problem instance.

The genetic learning paradigm was inspired by Darwinian natural selection and mutations in biological reproduction. New problem instances are tested against some sort of objective function in order to decide which already existing concepts survive.

The connectionist paradigm is also known as neural networks or parallel distributed systems. Some sort of learning algorithm, Boltzmann or back propagation for example are very well known ones, is used to calculate a credit assignment to every problem instance. Weights are readjusted with every evaluation of a new problem instance. The big difference between the connectionist approach and the three other approaches is that the connectionist approach evaluates a pattern of input in a holistic manner. Problem instances are represented as patterns of activation over a network consisting of simple elements that can only produce one type of output; they either produce output (fire) or they do not. These simple elements are called neurons [9].

3.6 Topic Maps

Topic maps provide a way of structuring semantic relations between information objects. They can be used to qualify content of information objects in order to provide some sort of categorization, to link objects to each other in order to enable efficient navigation, to create a certain view of a set of objects based on filtering and to structure a set of objects [11].

Topic maps consist of topics (which can denote any kind of object). An instance of such a topic is called an occurrence. The power of topic maps lies in the fact that topics play roles in associations. They are related to each other in a certain fashion. These relations are called associations and, like the topics themselves, they also have a name [10].

In section 3.4 we have introduced the notion of an ontology. An ontology is a kind of topic map where objects are occurrences, classes of objects are topics and their relations are associations.

3.7 SEMANTIC SCORER TYPES

The main function of the Semantic Scorers is to score all the connections between the source and target data model on a semantic level. In this project this scoring is limited to the history of previous projects. In the future more matching methods based on semantics can be added, such as value matching which was already mentioned in 2.3.2.2

Just like the Syntactic Scorers, the Semantic Scorers takes source and target attributes as input and returns a total score listing. Figure 20 depicts a graphic representation of the Semantic Scorers. Again the thesaurus is used to generate synonyms from the source attribute. After the connections between attributes have been created by adding new synonyms, these connections are mapped to the matches in the history.

The Semantic Scorer essentially works in the same way as a Syntactic. Figure 19 describes in more detail what happens in a Semantic Scorer. We will describe the algorithm used for history matching in more detail, as well as the algorithm for updating the history database.

Algorithm $SemanticScorer(source\_attribute, target\_attribute)$
1. ▷ Returns the overall semantic score of the connection between $source\_attribute$ and $target\_attribute$.
2. $source\_synonyms \leftarrow CreateSynonyms(source\_attribute)$
3. $score_1 \leftarrow \emptyset$
4. **for** each $synonym \in source\_synonyms$
5. **do**
6. $score_1 \leftarrow score_1 \cup \{Algorithm_1(synonym, target\_attribute)\}$
7. $score_2 \leftarrow \emptyset$
8. **for** each $synonym \in source\_synonyms$
9. **do**
10. $score_2 \leftarrow score_2 \cup \{Algorithm_2(synonym, target\_attribute)\}$
11. $score_3 \leftarrow \emptyset$
12. **for** each $synonym \in source\_synonyms$
13. **do**
14. $score_3 \leftarrow score_3 \cup \{Algorithm_3(synonym, target\_attribute)\}$
15.
16. ...
17.
18. $score \leftarrow \{\max(score_1), \max(score_2), \max(score_3), \ldots\}$
19. $total\_score \leftarrow avg(score)$
20. $transformations \leftarrow \emptyset$
21. $verified \leftarrow false$
22. $Vw \leftarrow \{wm, score, transformations, verified\}$
23. $Ew \leftarrow \{(wm, score), (wm, transformations), (wm, verified)\}$
24. $Aw \leftarrow \{(source\_attribute, wm), (wm, target\_attribute)\}$
25. $WeavingModel \leftarrow (Vw, Ew, Aw)$
26. **return** $(source\_attribute, WeavingModel, target\_attribute)$

Algorithm $CreateSynonyms$(name)
1. ▷ Returns a set of known synonyms of $name$
2. $synonymset \leftarrow \{name\}$
3. $synonymset \leftarrow synonymset \cup \{RemovePrefixes(name)\}$ $synonymset \leftarrow synonymset \cup \{TranslateAbbreviations(name)\}$ $synonymset \leftarrow synonymset \cup \{RemoveUnderscores(name)\}$
4. $thesaurusset \leftarrow \emptyset$
5. **for** each $element \in synonymset$
6. **do**
7. { let $Thesaurus(element)$ denote the set of synonyms that the thesaurus returns for $element$ }
8. $thesaurusset \leftarrow thesaurusset \cup Thesaurus(element)$
9. $synonymset \leftarrow synonymset \cup thesaurusset$
10. **return** $synonymset$

Figure 19: A pseudo code algorithm for the Semantic Scorer Types

In order to compare attribute connections to connections that were used in previous projects, there is need for some kind of mechanism to store connections used in previous projects. First of all, a representation is necessary

for these projects. Weaving models are used to represent the connections between source and target models. A more detailed description of how weaving models represent these connections is provided in section 3.7.1. Secondly, a representation for the history of projects is needed as well, in order to be able to match current connections to the weaving models already created in the past. The history is represented by an ontology, as is described in section 3.7.2. Finally a method is needed to compare current connections to connections stored in the history and a method to update the history once a new project is completed successfully. For this, machine learning techniques will be used. This is described in detail in section 3.7.3.

To achieve this we use a supervised learning algorithm, which is a machine learning technique, to create labeled mappings by feeding the algorithm training data. This training data should be precise and concise. This data is used to correctly classify future mappings.

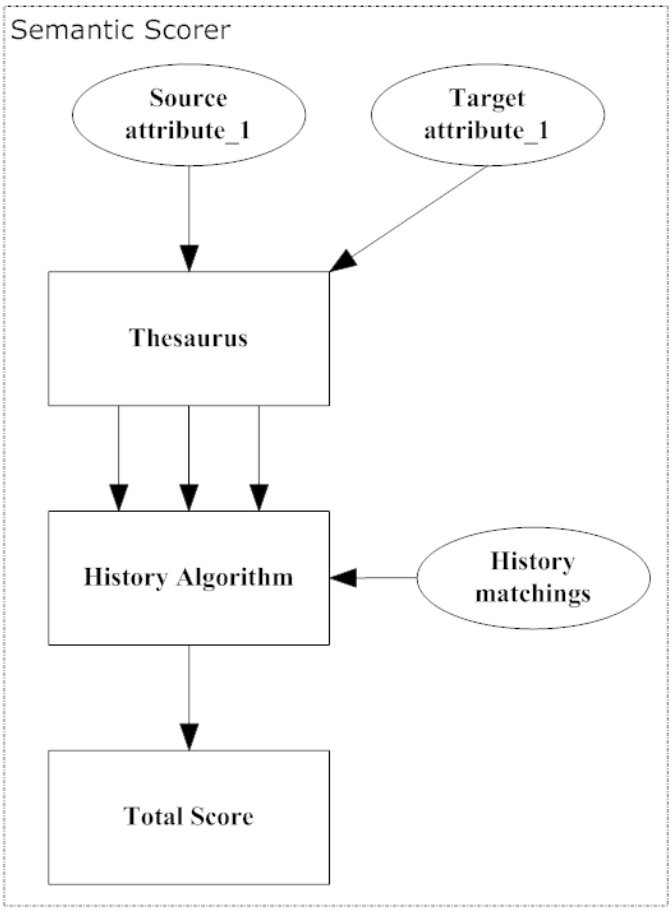

Figure 20: High level Semantic Scorer model

The history mappings are weighted; the higher a weight the more likely the stored connection in the history database was the correct one.

A stored history item is stored as a triple, containing three objects; the source attribute, the target attribute and a history information object.

$$\langle AS, AT, H \rangle$$

This history information object contains at least

- weight information, how relevant an attribute connection is within the history.
- a regular expression, representing values of source attribute in past projects.
- a transformation list, a list of required transformations to fit the source attribute to the target attribute.

To create better results in the future the history information object can be expanded.

A query for the Semantic Scorers inputs on the history will return a weight, if it exists. This weight is assigned to the attribute connection. The regular expression could be used to check the values from the source object; if the regular expression matches these values, the weight is added to the current weight of the attribute connection.

The result of the Semantic Scorer is a weighted weaving model. For each source attribute the connection with the highest weight should be the best option.

3.7.1 THE USE OF WEAVING MODELS

In section 2.2 the concept of weaving models was introduced. We have chosen to represent the connection between source and target data model by a weaving model because these connections contain metadata. A weaving model is a graph and it therefore supplies the ability to store metadata belonging to each connection. A connection between source attribute sa and target attribute ta is represented by the triple (sa, Wm, ta) where Wm is the weaving model. Wm is a directed graph consisting of a set of vertices Vw and a set of edges Ew and a set of associations Aw.

Vw is constructed as follows: for every connection there is a vertex representing that connection and a vertex for every type of metadata that we want to add to the weaving model.

Ew consists of an edge from the vertex representing a connection to every vertex representing a metadata field attached to that connection.

Aw is a set of associations connecting the attributes of the source model to the correct vertices in Wm and connecting the vertices in Wm to the correct attributes in the target model. There is an association between an attribute a in the source model and a vertex v in Wm if and only if a is connected to some attribute in the target model via the connection denoted by v – that is via the connection that contains the particular metadata that is attached to vertex v. Furthermore, there is an association between a vertex v in Wm and an attribute a in the target model if and only if there is a connection from some attribute in the source model to a via the connection denoted by v.

3.7.2 THE USE OF AN ONTOLOGY

For this project, the ontology as depicted in Figure 21 was constructed. The history matching algorithm uses this ontology to look up the certainty of a connection based on instantiations of this connection in previous projects. The certainty of connections in this ontology is updated after each successful completion of an integration project.

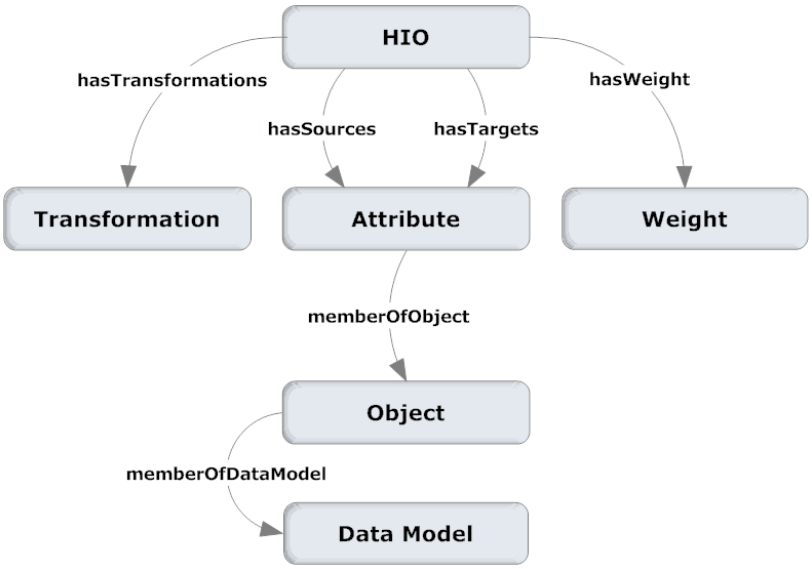

Figure 21: A History Information Object ontology representation

3.7.3 THE USE OF MACHINE LEARNING

Since the history is updated using approved weaving models, the learning is done inductively based on correct examples, no counterexamples are used. Counterexamples were not used since in the scope of this project they do not exist. Since any connection is possible there are no false connections. The inductive learning process is incremental. The set of weaving models which is the output of a completed integration project is used as a learning example for the history.

In updating the history, also a holistic connectionist approach [9] is used. As can be seen in the ontology that was designed for this project, a weaving model can be matched to a history information object by comparing not only source and target attributes, but also transformations, source and target object and data model. For our purposes a very simple updating rule is sufficient; if all of these properties of a weaving model match with the properties of a history information object, then the weights of this history information object are incremented by 1. Using this updating rule it is achieved that the more instances of a weaving model occur in integration projects, the higher the weight of the corresponding history information object will become.

Algorithm *UpdateHistory*(WMS)

1. ▷ Updates the history after approval of WMS as the correct set of weaving models for the current integration project.
2. **for** each $(sa, WM, ta) \in WMS$
3. **do**
4. { Let $WM = (Vw, Ew, Aw)$, $Aw = \{(sa, wm_1), (wm_1, ta)\}$, $sa \in DM_1$, $DM_1 = (V_1, E_1)$, $(sa, so) \in E_1$, $ta \in DM_2$, $DM_2 = (V_2, E_2)$, $(ta, to) \in E_2$ }
5. Find a HIO such that $hasScources(HIO, sa)$ and $hasTargets(HIO, ta)$ and $\forall_{trans,(wm_1,transformations)\in Ew \wedge (transformations,trans)\in Ew} hasTransformations(HIO, trans)$ and $memberOfObject(sa, so)$ and $memberOfObject(ta, to)$ and $memberOfDataModel(so, DM_1)$ and $memberOfDataModel(to, DM_2)$
6. **if** such a HIO exists
7. **then** $hasWeight(HIO, weight) \leftarrow hasWeight(HIO, weight + 1)$
8. **else** Create HIO with $hasScources(HIO, sa)$ and $hasTargets(HIO, ta)$ and $\forall_{trans,(wm_1,transformations)\in Ew \wedge (transformations,trans)\in Ew} hasTransformations(HIO, trans)$ and $memberOfObject(sa, so)$ and $memberOfObject(ta, to)$ and $memberOfDataModel(so, DM_1)$ and $memberOfDataModel(to, DM_2)$ and $hasWeigh(HIO, 1)$

Figure 22: A pseudo code algorithm for a history matching algorithm

3.8 Combining scorer results

Figure 11 shows how each scorer component is connected to the Weaving Model Comparer, which is the part of the system that grades the outcomes of the scorers and decides which weaving model should be presented to the user. The Score Comparer uses the output of the different scorers, the syntactic and the semantic scorer, to calculate a weighted average score for the connection between the source and the target attribute.

For example the output of a history scorer is more important than the object syntactic, because the history is more likely to produce a relevant score. Therefore its result should weigh more in calculating the combined weaving model.

Furthermore, using the scores and the transformations obtained from the semantic scorer, it creates a weaving model for source and target attribute that can be presented to the user in order to be verified manually.

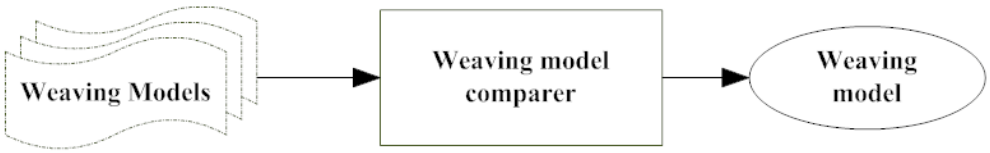

Figure 23: Weaving model comparer

The Weaving Model Comparer process will undergo the following steps to reach its goal:

- Remove the connections of all input weaving models with a score below a give threshold.
- Combine all remaining connections into one weaving model.
- Remove duplicate connections by evaluating which one has the highest score.

4 IMPLEMENTATION

This chapter describes the implementation details for the WMgen prototype. The WMgen prototype consists of different scorer types, that calculate a score based on for instance the structure (for a syntactic type scorer) or the semantics (for a semantic type scorer) of source and target model. This score is determined by the different algorithms that are executed in each scorer type. These scores are then combined and a single weaving model is generated. The design for this process was described in the previous chapter. This chapter describes how this design can be fit into an overall design of the WMgen component including conversion of XML input and output, data types, the use of a thesaurus, etc.

In section 1, communication with UDS is described. Section 2 describes the data types that were designed in order to construct the prototype. This section also provides details about XML input format, implementation of the history using RDF, the use of the EuroWordNet thesaurus and creation of synonyms based on translation procedures such as removing prefixes. Section 3 describes the architecture of the prototype. Dependencies of WMgen are given in section 4. Section 5 explains how to use WMgen. Setup and configuration are described in section 6.

4.1 UDS WMGEN PROTOTYPE

The WMgen will be part of the Enterprise Service Bus depicted in Figure 3. It will communicate with MDM Engine and MDM Client, both also depicted in Figure 3, via web services.

The Weaving Model Generator communicates with the MDM Client. It outputs a weaving model, which is visualized by the MDM Client as a transformation model with certainties attached to each transformation. The user is now able to correct and verify the model and save it to the metadata. The verified model is returned to the WMgen which uses it to update its history information.

The design of the Weaving Model Generator will be described in the rest of this chapter. The data model that was used will be described in section 4.2, section 4.2.3 describes the used thesaurus and in section 4.3 the architecture will be described. Finally, in section 4.5 and 4.6 the setup of the prototype will be explained.

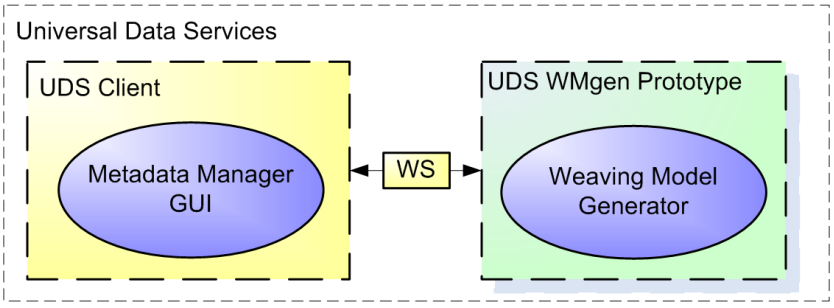

Figure 24: WMgen Setup

4.2 Data Types

In this section the data model of WMgen will be described. This data model consists of two separate models. DataModel is the class hierarchy for the XML input describing the datasources. ReturnModel is the class hierarchy for the output of scorer objects and WMgen itself.

4.2.1 Data Representation Models

4.2.1.1 DataModel

A data model consists of a set of objects and it has a name. An object consists of a set of attributes and it has a name. Finally, an attribute has a type and it also has a name. We choose to add an extra datastructure next to DataModel, DataModelObject and DataModelAttribute, as can be seen in Figure 25. This datastructure represents the name of each of these three datastructures. This provides a simple solution for creating and storing the synonyms of a name and the original name in case in early processing, prefixes were removed. A more detailed description of the DataModel datastructure is provided in the remainder of this section.

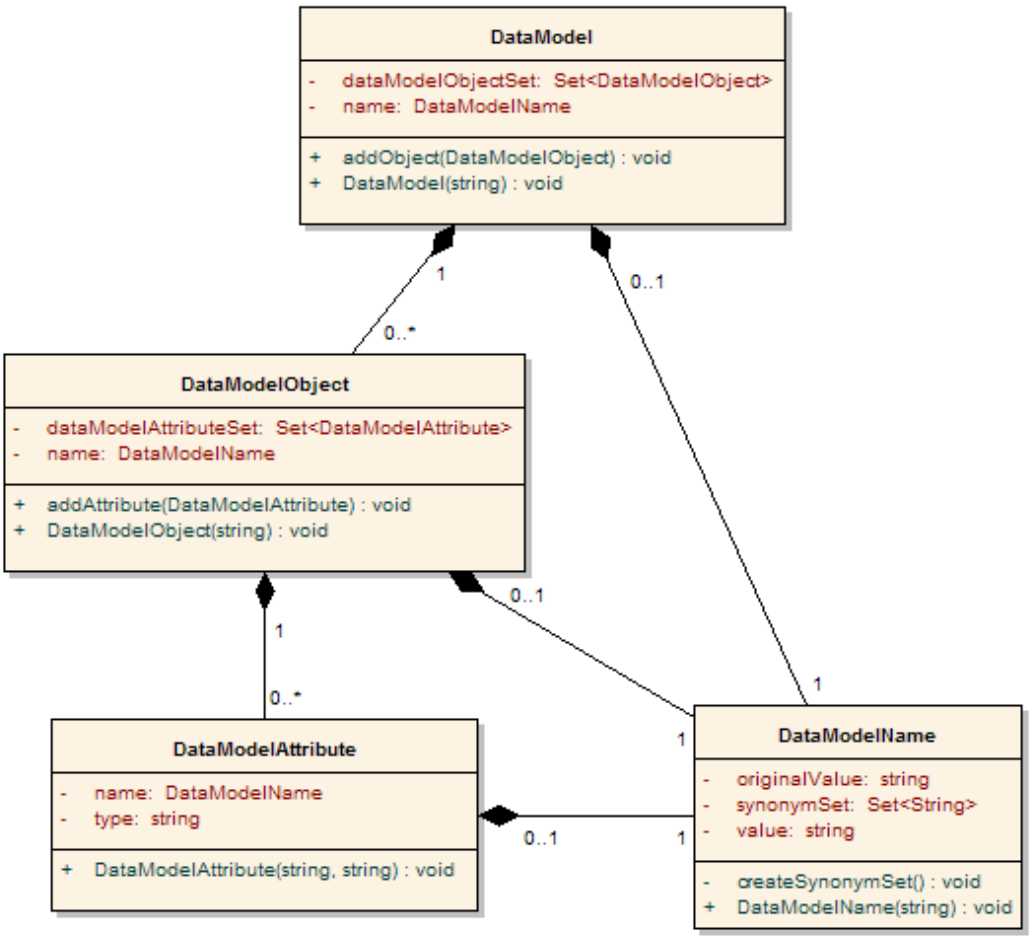

Figure 25: Class diagram for DataModel structure representation

4.2.1.2 RETURNMODEL

WMgen returns a set of weavingmodels. We represent a weaving model by the class Connection, as depicted in Figure 26. Such a connection has a score, which represents the certainty of the connections correctness. It also can have a TransformationList which defines how the source object should be transformed to fit in the target object. A weaving model, represented by the class Connection, has a source and a target, a score and a list of transformations.

The actual output of WMgen is an instantiation of the class AbstractReturnModel which, as mentioned above, consists of a set of instantiations of the Connections class. In designing the architecture of WMgen it was proven useful to also use this datastructure to represent intermediate score-calculations.

A minor alteration was necessary though, since also intermediate scores between objects are calculated. Therefore, two subclasses of AbstractReturnModel were created. The subclass ScoreModel is used to represent intermediate calculations which can be weavingmodels between either two attributes or two objects. The subclass WeavingModel now represents the actual output, which is a set of weavingmodels of which source and target are restricted to attributes.

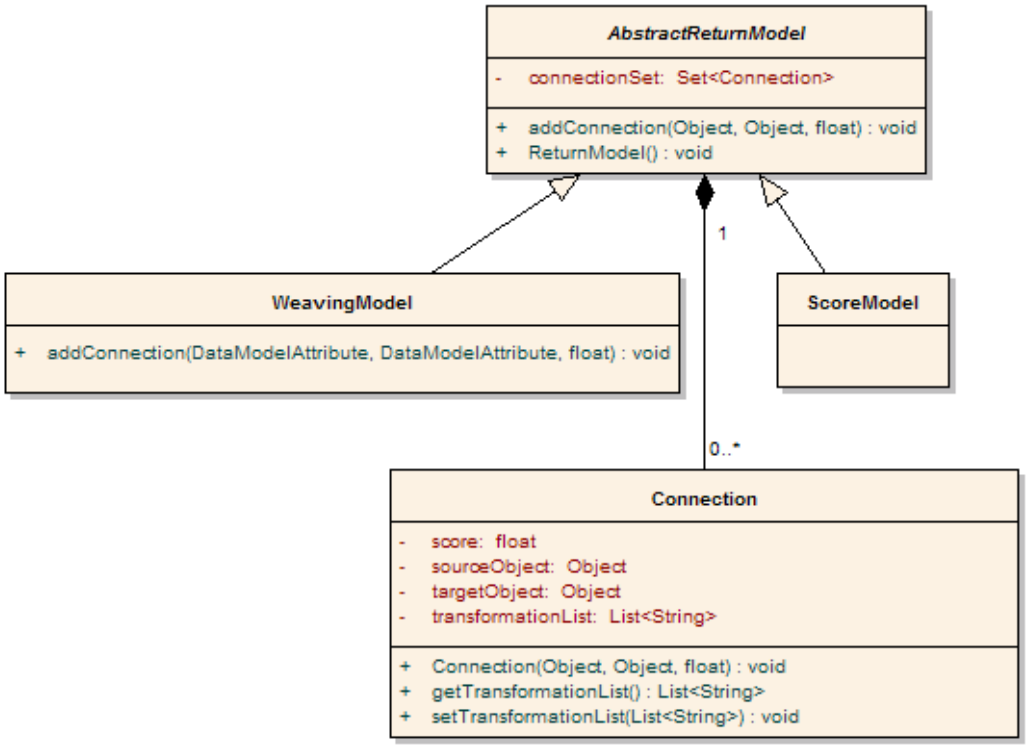

Figure 26: Class diagram for a ReturnModel (WeavingModel/ScoreModel)

4.2.2 MESSAGES

As part of the UDS Enterprise Service Bus WMgen communicates with the MDM and Sesame using messages. This section describes the global structure of these messages.

4.2.2.1 XML

The WMgen uses XML, describing the source and target data models, as input. The XML needs to be of a specific format in order for the WMgen to interpret it. The XML format follows the model as described in appendix B. The output that is created by the WMgen follows the WeavingModel as described in appendix C.

WMgen can also have XML as input that contains a weaving model. In this case there has been interaction with the user in order to update the weights of the existing history data. The input XML has the same format as the output XML from an execution of WMgen with source and target data models as input, except for the added field “verified” which contains a Boolean value. This value represents whether the user marked a connection as being correct. An example is given in appendix D. When WMgen is executed on an XML containing a weaving model, the history is updated accordingly and no output is generated.

4.2.2.2 History Information Object

History Information Objects (HIO) are part of the History Matcher, which stores a weighted weaving model. This semantic information is stored using a RDF database. The databases purpose is to support future semantic broadening of the History Information Object. WMgen uses the Sesame framework 16.

Each entry, a HIO, in the database contains source attribute and target attribute references and meta information like name and type. Beside a transformation list, the HIO also has a weight which is a counter. Each time a user verifies a transformation model using the UDI client, the weight of the corresponding HIO is incremented by 1. When matching new connections to the history, the algorithm evaluates the weight; the higher the weight, the more likely it is that its corresponding HIO represents the correct attribute transformation list for that new connection.

When a user has verified a transformation model, the model is returned to the WMgen and the HIOs are updated. The database thus contains a merger of all weaving models.

4.2.3 Thesaurus

WMgen uses a Thesaurus in order to be able to not only compare the actual names of attributes, objects and data models to each other, but also compare synonyms of pairs of names to each other. This will result in greater accuracy of scores for matchings. A Thesaurus delivers these synonyms to WMgen. WordNet is the most frequently used Thesaurus.

EuroWordNet is a Thesaurus that offers implementations of WordNet in some European languages, including Dutch. In this project the EuroWordNet Thesaurus is used, because URBIDATA works with municipalities in the Netherlands. In general these municipalities work with data models that use Dutch terms. If in the future WMgen will also need to be used for data models that use other languages, then language recognition software might be needed to recognize which WordNet variant should be used.

4.2.3.1 Translation procedures

Before the Thesaurus is used to look up synonyms for a model name, object name or attribute name, some translation procedures are executed in order to account for prefixes, underscores and abbreviations in a model. Prefixes are

substrings of *n* characters, for a certain natural number *n*, that are used as the first *n* characters for every name in a certain group. If for instance for a certain object, every attribute name starts with the substring “STR_”, then “STR_” is a prefix. Prefixes can be found in model names, object names and attribute names. When these prefixes are left out in the matching process, this could result in a more accurate score for this model. Therefore, before the thesaurus is used, a synonym is created for every name that possesses a prefix, consisting of all characters of the original name minus the prefix.

We can find prefixes as follows. Start with the first name of a group (for instance of a group of attributes). Compare this name to the second name and find out how many of the first characters of these two names are the same. Then compare the third name of this group to the substring that was find in the previous step and find out how many of the first characters of these two strings are the same. Continue with this process until all names are compared. The substring that is left over after the last comparison is a prefix for this group of models, objects or attributes.

Because all object and attribute names are necessary in order to find prefixes, removal of prefixes is done when creating the class model from the input XML (see section 4.3).

Abbreviations are defined only for target data models. For source data models there is no knowledge about the possible use of abbreviations, so defining any translations for abbreviations used in source models is not possible. For target models the used abbreviations are defined in the documentations of the standard that was used for the target model. These abbreviations and their meaning, their translation, need to be defined in the configuration file (see section 4.6 for more details on the syntax and use of the configuration file). A synonym is created for every name that contains an abbreviation, using the translation of the abbreviation in stead of the abbreviation.

4.3 ARCHITECTURE

The process of creating a weaving model is as follows. The algorithms are executed and their output scores are combined to an overall score of each scorer. Therefore classes that represent algorithms and classes that represent scorers are necessary. The output of the scores is combined to form the output weaving model, for this a class that represents the score combine procedure is needed.

Furthermore, which algorithms and scorers are used and in which order they are used is defined in a configuration file. This configuration forms the guideline according to which the WMgen component executes the process of forming a weaving model. A class is needed that represents the guidelines for this process as defined in the configuration file.

Finally WMgen needs to deal with XML input and output. For this classes are used that represent the translation procedures for XML input to the datamodel as depicted in 4.2.1.1 and for the created weaving model as depicted in 4.2.1.2 to XML output.

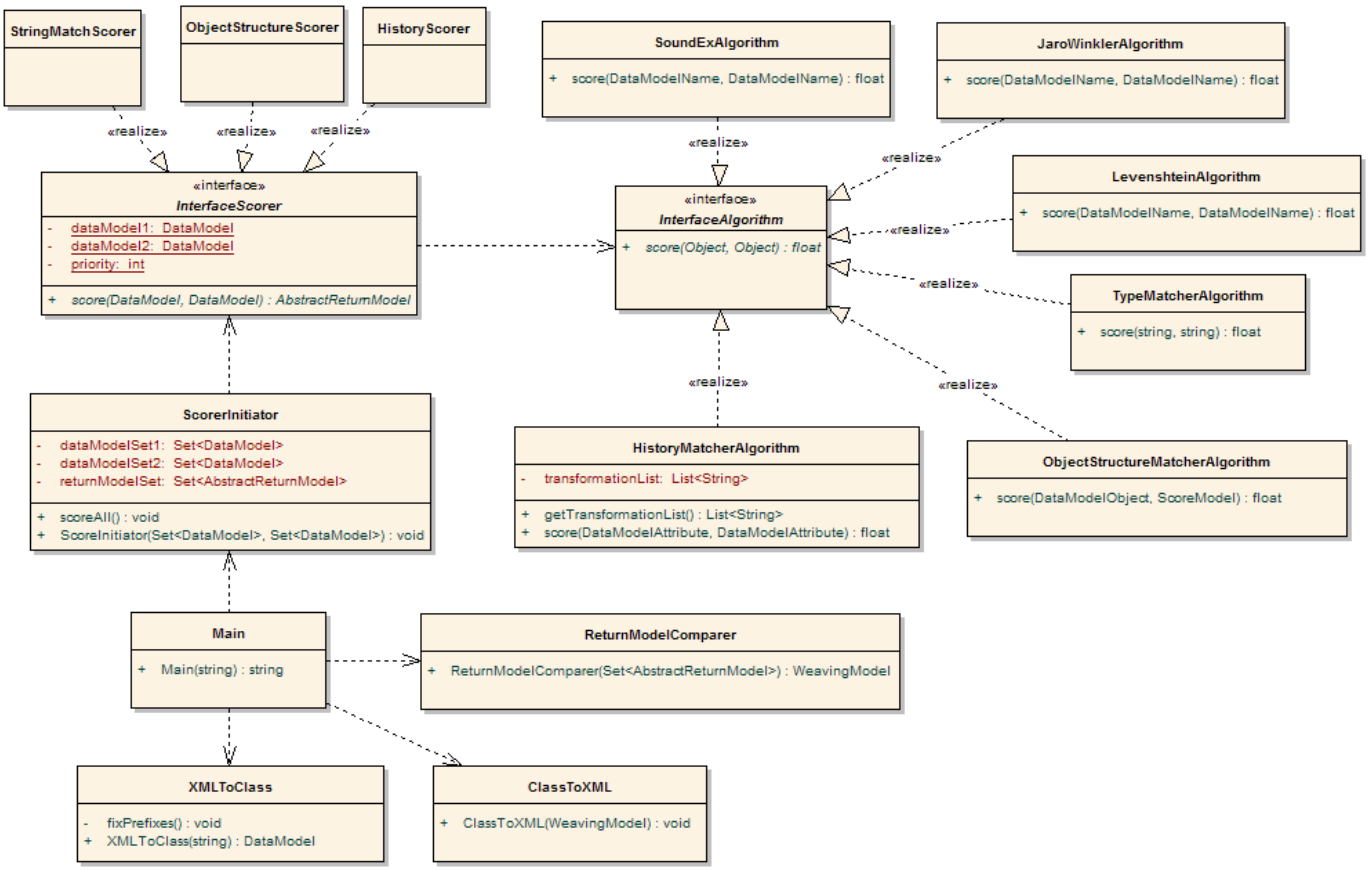

Figure 27: WMgen prototype system architecture

The resulting system architecture is depicted in Figure 27. The main method for the system is of course Main. The input is a filename containing an XML description of the source and target datamodels. These models are then parsed by XMLToClass and passed ScorerInitiator. The ScorerInitiator results in a set of ReturnModels, which are converted to a WeavingModel bij the ReturnModelComparer. This WeavingModel is passed to ClassToXML and converted back to a WeavingModel XML format.

The scores for a comparison are determined by the class InterfaceScorers. This class implements certain types of scorers, namely syntactic and semantic scorers. Each scorer is configured by the configuration file, that defines which algorithms should be used. Also each scorer has a priority. For example the StringMatchScorer should be executed before the StructureScorer, that uses the StringMatchScorer's results. The scorers used for the thesis project setting are described in 3.3 and 3.7. More types of scorers can be added in the future to generate better matchings.

A score is determined by algorithms that are described in this document. The algorithms are implemented in the child-classes of InterfaceAlgorithm. Each algorithm belongs to a scorer. An algorithm object determines the score for a comparison between two objects. These two objects can be for example a DataModel, a DataModelObject, a DataModelAttribute or even a string. The

algorithms used for the thesis project setting are described in 3.3.1. More algorithms can be added in the future.

Furthermore, the ScoreModels that are created by InterfaceScorer are compared in the class ReturnModelComparer. In this class, the optimal WeavingModel for the source- and targetmodels is determined.

Finally, the XMLToClass class implements an operation in order to generate a class model that can be used in the WMgen prototype and the ClassToXML class implements an operation in order to generate an XML file from the weaving model that was created by the WMgen prototype.

4.4 DEPENDENCIES

The Weaving Model Generator uses several external libraries. It uses libraries that implement matching algorithms, a Thesaurus and a repository of previously generated weaving models in order to enable history matching.

The implementations of all of the matching algorithms that are described in this document are available in the Simmetrics (uk.ac.shef.wit.simmetrics) library. Algorithms, not only matching algorithms from the Simmetrics library or other libraries, but also newly designed algorithms, can be added as an extension of the AbstractAlgorithm class in the future.

The Thesaurus that is used in the DataModelName class in order to generate synonyms is the Dutch variant of EuroWordNet.

The history matching algorithm uses a repository (as described in 4.2.2.2) of previously generated weaving models. The Sesame 2 library is needed in order to be able to communicate with this repository.

4.5 USING WMGEN

Now that we have explained the WMgen architecture all that is left to point out is how to operate the WMgen system. As described before the WMgen system needs an XML of the format described in section 4.2.2.1 as input. WMgen will then generate a weaving model that is to be presented to the Client Integrator. The Client Integrator will visualize the model. The user can then verify whether the model is correct by evaluating the scores of each transformation and the transformation itself. This verified model can then again be used as an input for WMgen which will store the verified model in the history repository.

For example one method for letting a user evaluate a transformation model is to present each transformation with a colored certainty (see Figure 28); the lower the certainty the more red the box will be and the higher the certainty the more green the box will be. Each certainty is accompanied with a checkbox. If a certainty is above a certain threshold, the checkbox is checked; this is usually the case. The user is now able to evaluate the transformation model; check a box to indicate that a transformation is accepted and uncheck a box otherwise. A good user would update the transformation accordingly and then send a 100% correct model to WMgen; this ensures better results from the HistoryMatcher.

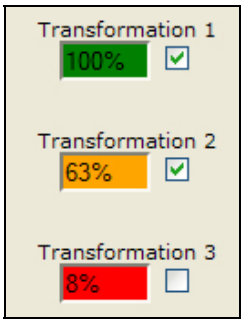

Figure 28: A MDM transformation model evaluation interface example

4.6 Setup and Configuration

The WMgen prototype is a stand-alone component. The user is able to pass XML data models as command line arguments to WMgen. The UDI must also use the WMgen component like this. In the future this might be done by event driven messaging, but for this prototype command line arguments suffice.

WMgen can be configured using a XML configuration file as depicted in Figure 29. The most important part of the configuration file describes which scorers should be used and which algorithms each scorer invokes. This way scorers and algorithms are easily added or removed from the weaving model generation process.

Furthermore each scorer has a priority. Scorers with a higher priority are executed before scorers with a lower priority. Priorities range from 1 to infinity; 1 being the highest priority.

This configuration file should be in the same directory as WMgen. Newly created scorers and algorithms should implement the *InterfaceScorer* and *InterfaceAlgorithm* classes, as described in section 4.3.

```
<?xml version="1.0" encoding="utf-8" ?>
<CONFIGURATION>
	<SCORERS>
	<SCORER CLASSNAME=”classname_1” PRIORITY=”1” WEIGHT=”1”>
		<USES>
			<ALGORITHM>
				<CLASSNAME>classname_2</CLASSNAME>
				<WEIGHT>weight_2</WEIGHT>
			</ALGORITHM>
			:
			:
			<ALGORITHM>
				<CLASSNAME>classname_n</CLASSNAME>
				<WEIGHT>weight_n</WEIGHT>
			</ALGORITHM>
		</USES>
	<SCORER>
	:
	:
	<SCORER CLASSNAME=”classname_3” PRIORITY=”4” WEIGHT=”1”>. . . .</SCORER>
	</SCORERS>
	<THRESHOLD>0.5</THRESHOLD>
</CONFIGURATION>
```

Figure 29: A WMgen configurationfile

5 PROTOTYPE RESULTS

In this section a test case is described that was used to test the units of the WMgen prototype that are relevant for this research project. Part of a transformation model created during a successfully completed UDS integration project [18] will be used. In section 5.1 the test case that is used throughout this chapter is described. Section 5.2 shows the output that should be expected. The used tests and their outcome are described in sections 5.3 through 5.8.

5.1 TEST CASE

Figure 30 depicts the partial target data model, taken from the UDS integration project for municipality of Westland, which was used for testing of the prototype. The target data model is named "[GIS] SA Adres" and it contains three objects that consist of several attributes. GIS is a reference to one of the standards mentioned in 2.1.2.

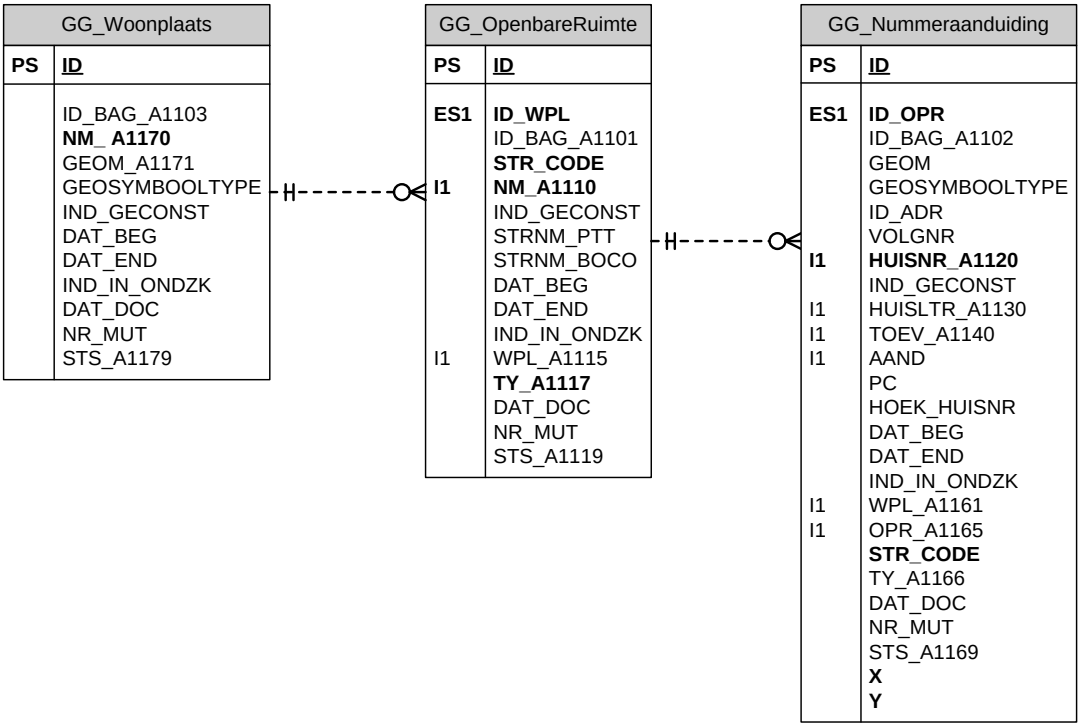

Figure 30: Partial target data model of the Westland GIS project

5.2 EXPECTED RESULTS

In Figure 31 the objects of the target data model (the right column) as well as the source data model (the left column) are listed. Figure 32 depicts part of the actual transformation model that was used in the Westland project for this part of the source and target model. These figures can be used as a reference for evaluating the performance of the prototype; they can be compared to the actual prototype results and serve as the expected results.

Figure 32 lists the attribute names below each name of the object that they are a part of. The object names are printed in bold. The object and attribute names of the source model are listed in the leftmost column, the object and attribute names of the target model are listed in the rightmost column and in the center column the transformations are listed.

[IMP] Adr4All	**[GIS] SA Adres**
ADR5_WOONPLAATS	SA_WOONPLAATS
ADR5_STRAAT	SA_OPENBARERUIMTE
ADR5_ADRESCYCLUS	SA_NUMMERAANDUIDING

Figure 31: Partial transformation model on an object level of the Westland GIS project

ADR5_WOONPLAATS		**SA_WOONPLAATS**
WOONPLKODE	=	ID_BAG_A1103
WOONPLBOCO	=	NM_A1170
DDINGANG	=	DAT_BEG
DDEINDE	=	DAT_END
ADR5_STRAAT		**SA_OPENBARERUIMTE**
STRAATKODE	=	STR_CODE
STRAATKODE	= “straat”	TY_A1117
STRAATBOCO	=	STRNM_BOCO
STRAAT_OFF	=	NM_A1110
STRAAT_PTT	=	STRNM_PTT
DDINGANG	=	DAT_BEG
DDEINDE	=	DAT_END
ADR5_ADRESCYCLUS		**SA_NUMMERAANDUIDING**
WOONPLKODE	=	WPL_A1161
STRAATKODE	=	STR_CODE
DDINGANG	=	DAT_BEG
DDEINDE	=	DAT_END
ADRESNR	=	ID_ADR
VOLGNR	=	VOLGNR
AAND	=	AAND
HUISNR	=	HUISNR_A1120
HUISLT	=	HUISLTR_A1130
TOEV	=	TOEV_A1140
POSTK_A	CONCAT	PC
POSTK_N		
X_KOORD	=	X
Y_KOORD	=	Y
VRY_VELD2	REPLACE(“,”; ”.”)	HOEK_HUISNR

Figure 32: Partial transformation model on attribute level of the Westland GIS project

5.3 SYNONYM CREATION

A few attributes are used from the test case described in the previous section to test the output of the synonym creation process.

5.3.1 INPUT

The input for this test case consists of the attributes depicted in Figure 33. Combinations of source and target attributes are used, because the lists of synonyms created for the source and the target attributes are expected to resemble each other. Testing these combinations then enables us to verify this expectation.

The source and target attributes STRAATKODE and STR_CODE were chosen because of the abbreviation “STR” and the underscore in “STR_CODE”. The combination DDEINDE and DAT_END was chosen in order to find out what happens to the abbreviation DD, which – like DAT - stands for date. Furthermore “EINDE” means END in Dutch. The POSTK_A, POSTK_N, PC combination is interesting because PC is an abbreviation for postal code, but POSTK is not. Finally, the combination WOONPLKODE, WPL_A1161 is interesting because “WPL” is an abbreviation, but “A1161” does not mean anything.

STRAATKODE	STR_CODE
DDEINDE	DAT_END
POSTK_A	PC
POSTK_N	
WOONPLKODE	WPL_A1161

Figure 33: Input attributes for synonym creation test case

5.3.2 RESULTS

The output for the synonyms generated by the DataModelName object as described in 4.2.1.1 is depicted in Figure 34 for the attributes used in the test case described in section 5.3.1. This output does not view the results generated using the thesaurus, because the outcome can not be predicted at this time as just the sample database for EuroWordNet is available.

Original attribute name	**Output**
STRAATKODE	STRAATKODE
STR_CODE	STRAATCODE
DDEINDE	DDEINDE
DAT_END	DATUMEND
POSTK_A	POSTKA
POSTK_N	POSTKN
PC	POSTCODE
WOONPLKODE	WOONPLKODE
WPL_A1161	WOONPLAATSA1161

Figure 34: Output for synonym creation

5.4 THE STRING MATCH SCORER

We use a few attributes from the test case described in section 5.1 to test the output of the string match scorer.

5.4.1 INPUT

The input for this test case consists of the attributes depicted in Figure 35.

Source attribute	Target attribute
STRAATKODE	STR_CODE
HUISNR	HUISNR_A1120
WOONPLBOCO	NM_A1170
STRAAT_PTT	STRNM_PTT

Figure 35: Input attributes for string matching test case

Again the running example STRAATKODE and STR_CODE is used. The other cases are chosen for the following reasons. The target of HUISNR has a postfix, WOONPLBOCO and NM_A1170 are completely different and STRAAT_PTT and STRNM_PTT will differ in the middle when NM is replaced by NAAM.

5.4.2 RESULTS

Since the string match scorer needs to output the average of the outcome of the Jaro Winkler algorithm described in section 3.3.1.1, the Levenshtein algorithm described in section 3.3.1.2 and the SoundEx algorithm described in section 3.3.1.3. Figure 36 depicts the outcome for each algorithm used by the string match scorer as well as the total average, which is the actual outcome of the scorer. This test was done as a unit test, therefore no abbreviations or synonyms were generated, the scores are solely based on the input strings.

Source attribute	Target attribute	Jaro Winkler	Levenshtein	Soundex	String match scorer
STRAATKODE	STR_CODE	0.8483	0.6	0.9667	0.8050
HUISNR	HUISNR_A1120	0.9333	0.5	1.0000	0.8111
WOONPLBOCO	NM_A1170	0.4083	0.0	0.5556	0.3213
STRAAT_PTT	STRNM_PTT	0.8448	0.7	0.9333	0.8260

Figure 36: String Match Scorer output for string matching test case

As a comparison we have depicted the results for the string match scorer as used in the final prototype, which uses abbreviations and synonyms in Figure 37.

We can clearly see higher scores for STRAATKODE, HUISNR and STRAAT_PTT. WOONPLBOCO has a slightly lower score. This can be ascribed to the fact that the strings already differ a lot, but when the NM abbreviation is replaced in NM_A1170 by NAAM, the strings differ 2 characters more than before.
This negative change is an anomaly however; even a human consultant would have trouble matching these two string without more information.

Source attribute	Target attribute	String match scorer
STRAATKODE	STR_CODE	0.879316
HUISNR	HUISNR_A1120	0.874074
WOONPLBOCO	NM_A1170	0.319753
STRAAT_PTT	STRNM_PTT	0.867711

Figure 37: String Match Scorer output using abbreviations and synonyms

5.5 The Object Structure Scorer

The objects from the test case described in section 5.1 are used to test the output of the syntactic. ScoreModels generated by scorers with a higher priority (in this case the string matcher scorer) are used in as input for the syntactic algorithm.

5.5.1 Results

Figure 38 shows the scores between each object pair. The scores are calculated by taking the number of attribute scores *na* from the input scoremodels above a certain threshold (in this case 0.5), dividing this number by the total number of child attributes *ta* in the two objects and finally multiplying it with the string match *sm* between the two object strings using the same algorithms (3.3.1.1-3.3.1.3) as the string match scorer.

i.e. ADR5_WOONPLAATS and SA_WOONPLAATS is calculated as follows:

$$\frac{na}{ta} \cdot sm = \frac{6}{16} \cdot 0.90769225 = 0.340385$$

Source Object	Target Object	String Match	Nr above threshold	Score
ADR5_WOONPLAATS	SA_WOONPLAATS	0.90769225	6	**0.340385**
ADR5_WOONPLAATS	SA_OPENBARERUIMTE	0.58464056	6	0.175392
ADR5_WOONPLAATS	SA_NUMMERAANDUIDING	0.51637430	6	0.103275
ADR5_STRAAT	SA_WOONPLAATS	0.55788660	10	0.309937
ADR5_STRAAT	SA_OPENBARERUIMTE	0.52168750	22	**0.521688**
ADR5_STRAAT	SA_NUMMERAANDUIDING	0.56485910	14	0.247126
ADR5_ADRESCYCLUS	SA_WOONPLAATS	0.49903846	9	0.166346
ADR5_ADRESCYCLUS	SA_OPENBARERUIMTE	0.38888893	12	0.150538
ADR5_ADRESCYCLUS	SA_NUMMERAANDUIDING	0.38888893	28	**0.265583**

Figure 38: Output for object structure test case

The scores generated by the object structure scorer seem to be low. However we can clearly see that the highest scores also resemble the correct object connections.

5.6 THE SEMANTIC SCORER

A few attributes from the test case described in section 5.1 are used to test the output of the semantic scorer.

5.6.1 INPUT

Before the testing starts, we make sure that the outcome of the test case is stored in the history. Now, if we try to match the attributes listed in Figure 39, it is expected that the correct transformations are returned. These transformations are described in the results section.

Source attribute	**Target attribute**
STRAATKODE	TY_A1117
POSTK_A	PC
POSTK_N	
VRY_VELD2	HOEK_HUISNR

Figure 39: Input attributes for semantic scorer test case

5.6.2 RESULTS

The transformations that are listed in Figure 40 are the results of the semantic scorer. The score values seem to be irrelevant, because they always equal 1. However this is to be expected as the user has verified these connections in the past to be correct. The scores are used to increase the total score of a connection in the ReturnModelComparer component of the prototype.

Source attribute	**Target attribute**	**Transformation**	**Score**
STRAATKODE	TY_A1117	= “straat”	1
POSTK_A	PC	CONCAT	1
POSTK_N			
VRY_VELD2	HOEK_HUISNR	REPLACE(“,”; ”.”)	1

Figure 40: Output for semantic scorer test case

5.7 THE SCORE COMPARER

5.7.1 INPUT

The set score models generated from the various scorers as depicted in Figure 37, Figure 38 and Figure 40 is the input of the ReturnModelComparer.
The ReturnModelComparer first filters all connections below a certain threshold (given by the user in a configuration file), then merges the remaining connections into one weaving model and finally removes duplicate connections.

5.7.2 RESULTS

Figure 41 depicts the final result using a threshold of 0.5 and weights of 1. The weights with value 1 mean that each algorithm and scorer is considered equally capable. Also the threshold makes the score comparer drop results with a score lower than 0.5; however the resulting weaving model may still contain lower results than 0.5, because these values are calculated from the remaining score connections.

If this weaving model is compared against the expected model depicted in 5.2, some mistakes are revealed; in Figure 41 these are printed italic. These mistakes can be categorized in four types of wrong results:

- Connection seems ok at first, but is actually wrong. For example ADRESNR and HOEK_HUISNR are very similar. A human could easily make the same mistake.
- Connections that have similar properties easily can be wrongfully outputted, because they also result in similar scores. For example DDINGANG and DAT_BEG/DAT_END.
- Mismatches. For example STRAAT_OFF and STR_CODE.
- Missing results. For example WOONPLKODE and ID_BAG_A1103.

User evaluation of the resulting weaving model, checking its correctness and updating it accordingly, should eliminate these mistakes in future runs. The history will become more detailed, whereas these prototype runs were executed with a history containing four history objects. Also better semantic evaluations of the data, using for example value matching as mentioned in section 2.3.2.2, will improve performance.

Source Attribute	Target Attribute	Transformations	Score
ADR5_ADRESCYCLUS.AAND	SA_NUMMERAANDUIDING.AAND		0.632791
ADR5_ADRESCYCLUS.ADRESNR	*SA_NUMMERAANDUIDING.HOEK_HUISNR*		*0.448725*
ADR5_ADRESCYCLUS.DDEINDE	SA_NUMMERAANDUIDING.DAT_END		0.509180
ADR5_ADRESCYCLUS.DDINGANG	*SA_NUMMERAANDUIDING.DAT_END*		*0.537463*
ADR5_ADRESCYCLUS.HUISLT	SA_NUMMERAANDUIDING.HUISLTR_A1130		0.498532
ADR5_ADRESCYCLUS.HUISNR	SA_NUMMERAANDUIDING.HUISNR_A1120		0.569828
ADR5_ADRESCYCLUS.POSTK_A	SA_NUMMERAANDUIDING.PC	CONCAT	0.622298
ADR5_ADRESCYCLUS.POSTK_N	SA_NUMMERAANDUIDING.PC	CONCAT	0.622298
ADR5_ADRESCYCLUS.STRAATKODE	SA_NUMMERAANDUIDING.STR_CODE		0.572449
ADR5_ADRESCYCLUS.TOEV	SA_NUMMERAANDUIDING.TOEV_A1140		0.512421
ADR5_ADRESCYCLUS.VOLGNR	SA_NUMMERAANDUIDING.VOLGNR		0.632791
ADR5_ADRESCYCLUS.VRY_VELD2	SA_NUMMERAANDUIDING.HOEK_HUISNR	REPLACE(","; ".")	0.632791
ADR5_ADRESCYCLUS.X_KOORD	SA_NUMMERAANDUIDING.X		0.392977
ADR5_ADRESCYCLUS.Y_KOORD	SA_NUMMERAANDUIDING.Y		0.392977
ADR5_STRAAT.DDEINDE	SA_OPENBARERUIMTE.DAT_END		0.637233
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_BEG		0.665515
ADR5_STRAAT.STRAAT_OFF	*SA_OPENBARERUIMTE.STR_CODE*		*0.675473*
ADR5_STRAAT.STRAAT_PTT	SA_OPENBARERUIMTE.STRNM_PTT		0.694699
ADR5_STRAAT.STRAATBOCO	SA_OPENBARERUIMTE.STRNM_BOCO		0.701747
ADR5_STRAAT.STRAATKODE	SA_OPENBARERUIMTE.TY_A1117	="straat"	0.760844
ADR5_WOONPLAATS.DDEINDE	SA_WOONPLAATS.DAT_END		0.546581
ADR5_WOONPLAATS.DDINGANG	*SA_WOONPLAATS.DAT_END*		*0.574864*

Figure 41: Output weaving model for the score comparer with threshold 0.5

5.8 VARYING TESTPARAMETERS

WMgen allows a number of parameters to be tweaked to reach the configuration that generates the best results. Scorers and Algorithms can be added/removed using a configuration file. Also each Scorer/Algorithm can be given a weight; this weight indicates the importance of a Scorer/Algorithm. Finally there is a threshold parameter; this parameter is used to remove scores that are lower than the threshold.

5.8.1 ADDING/REMOVING SCORERS AND ALGORITHMS

By varying the scorers in the configuration file we can adjust WMgen to try and generate better results. Figure 42 depicts the results of an analysis of these variations. The table is structured as follows. The top rows that are printed in bold are correct attribute connections as stated in Figure 32. The bottom rows which are not bold, are incorrect connections. When a cell contains a **–** symbol, it means that the resulting weaving model did not contain that attribute connection.

We define:

- The number of incorrect connections: $\langle \#\neg bold : score > 0 \rangle$
- The number of missing connections: $\langle \#bold : score = "-" \rangle$
- The number of correct connections: $\langle \#bold : score > 0 \rangle$

When taking into account that String Match should have priority over Object Structure and Object Structure over History[1], the following 4 scorer combinations can be tested:

- String Match
- String Match – Object Structure
- String Match – History
- String Match – Object Structure – History

We are now able to count the number of faulty results, the number of missing correct results and the number of correct results per combination. By dividing the number of incorrect results by the number of correct results we get a measure for identifying the best configuration. The lower this score, the better; it means that there are a lot less faulty than correct generated connections.

$$measure = \frac{\#incorrect + \#missing}{\#total}$$

The String Match – Object Structure scorer combination got the best score 0.217391. However if we evaluate the individual scores of the second result (String Match – Object Structure – History), we see that this combination of scorers has a larger number of unique source attributes.

ADR5_ADRESCYCLUS.VRY_VELD2 ←→ A_NUMMERAANDUIDING.HOEK_HUISNR is a unique connection generated by the History Scorer, while

[1] The String Match Scorer should have priority over the Object Structure Scorer, because the Object Structure Scorer uses the weaving model generated by higher priority Scorers (in this case the String Match Scorer).

ADR5_ADRESCYCLUS.DDINGANG ←→ SA_NUMMERAANDUIDING.DAT_BEG is correctly identified by the SO combination, but missing in the SOH combination. The missing connection is of less importance, because these attributes are apparent in more than one object and are therefore more likely to be matched correctly on other occasions. Also the History Scorer uses learning algorithms and could learn that ADR5_ADRESCYCLUS.DDINGANG ←→ SA_NUMMERAANDUIDING.DAT_BEG is actually a correct match; the results of the String Match Scorer and Object Structure Scorer are not capable of generating the ADR5_ADRESCYCLUS.VRY_VELD2 ←→ A_NUMMERAANDUIDING.HOEK_HUISNR connection in future runs. Therefore the SOH combination probably is a better choice. This result proves that adding more scorers might improve the scores even more.

		SO	SOH	SH	S
ADR5_ADRESCYCLUS.AAND	**SA_NUMMERAANDUIDING.AAND**	0,632791	0,632791	1	1
ADR5_ADRESCYCLUS.DDEINDE	**SA_NUMMERAANDUIDING.DAT_END**	0,50918	0,50918	-	-
ADR5_ADRESCYCLUS.DDINGANG	**SA_NUMMERAANDUIDING.DAT_BEG**	0,537463	-	0,809343	-
ADR5_ADRESCYCLUS.HUISLT	**SA_NUMMERAANDUIDING.HUISLTR_A1130**	0,498532	0,498532	0,731481	0,731481
ADR5_ADRESCYCLUS.HUISNR	**SA_NUMMERAANDUIDING.HUISNR_A1120**	0,569828	0,569828	0,874074	0,874074
ADR5_ADRESCYCLUS.POSTK_A	**SA_NUMMERAANDUIDING.PC**	0,611804	0,622298	0,979012	0,958025
ADR5_ADRESCYCLUS.POSTK_N	**SA_NUMMERAANDUIDING.PC**	0,611804	0,622298	0,979012	0,958025
ADR5_ADRESCYCLUS.STRAATKODE	**SA_NUMMERAANDUIDING.STR_CODE**	0,572449	0,572449	-	-
ADR5_ADRESCYCLUS.TOEV	**SA_NUMMERAANDUIDING.TOEV_A1140**	0,512421	0,512421	0,759259	0,759259
ADR5_ADRESCYCLUS.VOLGNR	**SA_NUMMERAANDUIDING.VOLGNR**	0,632791	0,632791	1	1
ADR5_ADRESCYCLUS.VRY_VELD2	**SA_NUMMERAANDUIDING.HOEK_HUISNR**	-	0,632791	1	-
ADR5_ADRESCYCLUS.X_KOORD	**SA_NUMMERAANDUIDING.X**	0,392977	0,392977	0,52037	0,52037
ADR5_ADRESCYCLUS.Y_KOORD	**SA_NUMMERAANDUIDING.Y**	0,392977	0,392977	0,52037	0,52037
ADR5_STRAAT.DDEINDE	**SA_OPENBARERUIMTE.DAT_END**	0,637233	0,637233	-	0,752778
ADR5_STRAAT.DDINGANG	**SA_OPENBARERUIMTE.DAT_BEG**	0,665515	0,665515	-	-
ADR5_STRAAT.STRAAT_PTT	**SA_OPENBARERUIMTE.STRNM_PTT**	0,694699	0,694699	0,867711	0,867711
ADR5_STRAAT.STRAATBOCO	**SA_OPENBARERUIMTE.STRNM_BOCO**	0,701747	0,701747	0,881807	0,881807
ADR5_STRAAT.STRAATKODE	**SA_OPENBARERUIMTE.TY_A1117**	-	0,760844	1	-
ADR5_STRAAT.STRAATKODE	**SA_OPENBARERUIMTE.STR_CODE**	0,700502	-	-	0,879316
ADR5_WOONPLAATS.DDEINDE	**SA_WOONPLAATS.DAT_END**	0,546581	0,546581	0,752778	-
ADR5_ADRESCYCLUS.ADRESNR	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,448725	0,448725	0,631868	0,631868
ADR5_ADRESCYCLUS.DDEINDE	SA_WOONPLAATS.DAT_END	-	-	0,752778	-
ADR5_ADRESCYCLUS.DDEINDE	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,752778
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_END	-	0,537463	-	-
ADR5_ADRESCYCLUS.DDINGANG	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,809343
ADR5_ADRESCYCLUS.STRAATKODE	SA_OPENBARERUIMTE.STR_CODE	-	-	0,879316	0,879316
ADR5_STRAAT.DDEINDE	SA_WOONPLAATS.DAT_END	-	-	0,752778	-
ADR5_STRAAT.DDINGANG	SA_NUMMERAANDUIDING.DAT_BEG	-	-	0,809343	-
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,809343
ADR5_STRAAT.STRAAT_OFF	SA_OPENBARERUIMTE.STR_CODE	0,675473	0,675473	0,829259	0,829259
ADR5_WOONPLAATS.DDEINDE	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,752778
ADR5_WOONPLAATS.DDINGANG	SA_NUMMERAANDUIDING.DAT_BEG	-	-	0,809343	-
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_END	0,574864	0,574864	-	-
ADR5_WOONPLAATS.DDINGANG	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,809343
	Number wrong	3	4	7	8
	Number missed	2	2	5	7
	Number missed & wrong	5	6	12	15
	Number right	18	18	15	13
	missed + wrong : all	**0,217391**	**0,25**	**0,444444**	**0,535714**

Figure 42: Variation test for Scorers

We can also vary the algorithms in the configuration file and try to generate better results. Figure 43 depicts the results of an analysis of variations on the algorithms used by the String Match Scorer. We now find the following algorithm combinations that can be tested:

- Jaro-Winkler
- Levenshtein
- SoundEx
- Jaro-Winkler – Levenshtein

- Jaro-Winkler – SoundEx
- Levenshtein – SoundEx
- Jaro-Winkler – Levenshtein - SoundEx

Again we are now able to count the number of faulty results, the number of missing correct results and the number of correct results per combination.

The combination of all algorithms generates the best results with an analysis score of 0.28. Again this result proves that adding more scorers might improve the scores even more.

		JLS	JS	LS	J	JL	L	S
ADR5_ADRESCYCLUS.STRAATKODE	**SA_NUMMERAANDUIDING.STR_CODE**	0,572449	0,927467	0,541641	0,681493	0,52330834	0,393579	1,889567
ADR5_ADRESCYCLUS.AAND	**SA_NUMMERAANDUIDING.AAND**	0,632791	0,941057	0,618564	0,708672	0,61382115	0,547426	1,889567
ADR5_ADRESCYCLUS.DDEINDE	**SA_NUMMERAANDUIDING.DAT_END**	0,50918	-	0,462314	0,650339	0,42840448	-	1,889567
ADR5_ADRESCYCLUS.DDINGANG	**SA_NUMMERAANDUIDING.DAT_BEG**	-	-	-	-	-	0,320153	-
ADR5_ADRESCYCLUS.HUISLT	**SA_NUMMERAANDUIDING.HUISLTR_A1130**	0,498532	0,906335	0,451897	0,639228	0,41243225	-	1,889567
ADR5_ADRESCYCLUS.HUISNR	**SA_NUMMERAANDUIDING.HUISNR_A1120**	0,569828	0,929946	0,53523	0,68645	-	-	1,889567
ADR5_ADRESCYCLUS.POSTK_A	**SA_NUMMERAANDUIDING.PC**	0,622298	0,939205	0,604675	0,704968	0,5980804	0,519648	1,889567
ADR5_ADRESCYCLUS.POSTK_N	**SA_NUMMERAANDUIDING.PC**	0,622298	0,939205	0,604675	0,704968	0,5980804	0,519648	1,889567
ADR5_ADRESCYCLUS.TOEV	**SA_NUMMERAANDUIDING.TOEV_A1140**	0,512421	0,89939	0,465786	0,653117	0,44715446	-	1,861789
ADR5_ADRESCYCLUS.VOLGNR	**SA_NUMMERAANDUIDING.VOLGNR**	0,632791	0,941057	0,618564	0,708672	0,61382115	0,547426	1,889567
ADR5_ADRESCYCLUS.VRY_VELD2	**SA_NUMMERAANDUIDING.HOEK_HUISNR**	0,632791	0,941057	0,618564	0,708672	0,61382115	0,547426	1,778456
ADR5_ADRESCYCLUS.WOONPLKODE	**SA_NUMMERAANDUIDING.WPL_A1161**	-	0,789668	0,371341	-	-	-	1,861789
ADR5_ADRESCYCLUS.X_KOORD	**SA_NUMMERAANDUIDING.X**	0,392977	0,789668	-	0,583672	-	-	-
ADR5_ADRESCYCLUS.Y_KOORD	**SA_NUMMERAANDUIDING.Y**	0,392977	0,789668	-	0,583672	-	-	-
ADR5_STRAAT.DDEINDE	**SA_OPENBARERUIMTE.DAT_END**	0,637233	1,122943	0,604594	0,892215	0,551714	-	-
ADR5_STRAAT.DDINGANG	**SA_OPENBARERUIMTE.DAT_BEG**	0,665515	1,122753	0,647207	-	0,59413826	-	1,638227
ADR5_STRAAT.STRAAT_PTT	**SA_OPENBARERUIMTE.STRNM_PTT**	0,694699	1,129816	0,683921	0,905961	0,6379142	0,42915	1,638227
ADR5_STRAAT.STRAATBOCO	**SA_OPENBARERUIMTE.STRNM_BOCO**	0,701747	1,134893	0,689415	0,916116	0,648486	0,440139	-
ADR5_STRAAT.STRAATKODE	**SA_OPENBARERUIMTE.TY_A1117**	0,760844	1,152109	0,760844	0,950548	0,7371307	0,582996	-
ADR5_WOONPLAATS.DDEINDE	**SA_WOONPLAATS.DAT_END**	0,546581	1,123237	0,513942	0,810417	0,48477563	-	-
ADR5_WOONPLAATS.DDINGANG	**SA_WOONPLAATS.DAT_BEG**	-	1,123048	0,556556	0,810038	0,52719986	0,329458	1,662981
ADR5_ADRESCYCLUS.ADRESNR	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,448725	0,789958	0,41023	0,573141	0,42105561	0,297425	-
ADR5_ADRESCYCLUS.ADRESNR	SA_NUMMERAANDUIDING.AAND	-	-	-	-	-	-	1,756233
ADR5_ADRESCYCLUS.DDEINDE	SA_WOONPLAATS.DAT_END	-	0,942147	-	-	-	-	-
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_END	0,537463	-	0,504927	0,64996	0,4708287	-	1,889567
ADR5_ADRESCYCLUS.DDINGANG	SA_WOONPLAATS.DAT_BEG	-	0,941958	-	-	-	-	-
ADR5_ADRESCYCLUS.HUISNR	SA_NUMMERAANDUIDING.HOEK_HUISNR	-	-	-	-	0,520964	0,404568	-
ADR5_ADRESCYCLUS.WOONPLKODE	SA_NUMMERAANDUIDING.STR_CODE	-	-	-	0,475339	-	-	-
ADR5_ADRESCYCLUS.X_KOORD	SA_NUMMERAANDUIDING.PC	-	-	-	-	-	-	1,736789
ADR5_ADRESCYCLUS.Y_KOORD	SA_NUMMERAANDUIDING.PC	-	-	-	-	-	-	1,736789
ADR5_STRAAT.DDEINDE	SA_OPENBARERUIMTE.DAT_BEG	-	-	-	-	-	-	1,638227
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_END	-	-	-	0,891836	-	0,355723	-
ADR5_STRAAT.STRAAT_OFF	SA_OPENBARERUIMTE.STR_CODE	0,675473	1,128221	0,656677	0,902771	0,6090751	0,374662	-
ADR5_STRAAT.STRAAT_OFF	SA_OPENBARERUIMTE.STRNM_PTT	-	-	-	-	-	-	1,638227
ADR5_STRAAT.STRAATBOCO	SA_OPENBARERUIMTE.STRNM_PTT	-	-	-	-	-	-	1,638227
ADR5_STRAAT.STRAATKODE	SA_OPENBARERUIMTE.STRNM_PTT	-	-	-	-	-	-	1,638227
ADR5_WOONPLAATS.DDEINDE	SA_WOONPLAATS.DAT_BEG	-	-	-	-	-	-	1,662981
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_END	0,574864	-	-	-	-	-	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_WOONPLAATS.IND_GEOCONST	-	0,914332	-	-	-	-	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_OPENBARERUIMTE.STRNM_BOCO	-	-	-	0,528764	-	-	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_WOONPLAATS.GEOM_A1171	-	-	-	-	-	-	1,496314
ADR5_WOONPLAATS.WOONPLBOCO	SA_OPENBARERUIMTE.WPL_A1115	-	-	0,369706	-	-	-	-
ADR5_WOONPLAATS.WOONPLKODE	SA_WOONPLAATS.IND_IN_ONDZK	-	0,929487	-	-	-	-	-
ADR5_WOONPLAATS.WOONPLKODE	SA_OPENBARERUIMTE.STR_CODE	-	-	-	0,485907	-	-	-
ADR5_WOONPLAATS.WOONPLKODE	SA_WOONPLAATS.IND_GEOCONST	-	-	-	-	-	-	1,496314
ADR5_WOONPLAATS.WOONPLKODE	SA_OPENBARERUIMTE.WPL_A1115	-	-	0,369706	-	-	-	-
	Number wrong	4	6	5	7	4	4	11
	Number missed	3	2	3	3	5	10	7
	Number missed & wrong	7	8	8	10	9	14	18
	Number right	18	19	18	18	16	11	14
	missed + wrong : all	**0,28**	**0,296296**	**0,307692**	**0,357143**	**0,36**	**0,56**	**0,5625**

Figure 43: Variation test for String Match algorithms

5.8.2 THRESHOLD

The threshold is used to filter results with a lower score. Again we use a similar analysis procedure and generate results for threshold values 0.1 through 0.9. Figure 44 depicts the results of this analysis. It can be concluded that a threshold of 0.5 generates the best results, closely followed by a threshold of 0.7 and 0.6.

		0,5	0,7	0,6	0,2	0,4	0,3	0,8	0,1	0,9
ADR5_ADRESCYCLUS.AANI	SA_NUMMERAANDUIDING.AANI	0,632791	0,575881	0,599594	1,76626	0,798781	1,410569	0,53794	2,344851	0,51897
ADR5_ADRESCYCLUS.DDEINDE	SA_NUMMERAANDUIDING.DAT_END	0,50918	0,45227	0,475982	1,642649	-	1,286958	-	2,22124	-
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_BEG	0,537463	0,480552	0,504265	-	-	-	0,442612	-	-
ADR5_ADRESCYCLUS.HUISLT	SA_NUMMERAANDUIDING.HUISLTR_A113(	0,498532	0,441621	0,465334	1,632001	0,664521	1,27631	-	2,210592	-
ADR5_ADRESCYCLUS.HUISNR	SA_NUMMERAANDUIDING.HUISNR_A112(	0,569828	0,512918	0,536631	1,703297	0,735818	1,347606	0,474977	2,281888	-
ADR5_ADRESCYCLUS.POSTK_A	SA_NUMMERAANDUIDING.PC	0,622298	0,565387	0,5891	1,755766	0,788287	1,400075	0,527447	2,334357	0,508476
ADR5_ADRESCYCLUS.POSTK_N	SA_NUMMERAANDUIDING.PC	0,622298	0,565387	0,5891	1,755766	0,788287	1,400075	0,527447	2,334357	0,508476
ADR5_ADRESCYCLUS.STRAATKODE	SA_NUMMERAANDUIDING.STR_CODE	0,572449	0,515539	0,539252	1,705918	0,738439	1,350227	0,477599	2,284509	-
ADR5_ADRESCYCLUS.TOEV	SA_NUMMERAANDUIDING.TOEV_A114(	0,512421	0,45551	0,479223	1,64589	0,67841	1,290199	-	2,224481	-
ADR5_ADRESCYCLUS.VOLGNR	SA_NUMMERAANDUIDING.VOLGNR	0,632791	0,575881	0,599594	1,76626	0,798781	1,410569	0,53794	2,344851	0,51897
ADR5_ADRESCYCLUS.VRY_VELD2	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,632791	0,575881	0,599594	1,595791	0,798781	1,2401	0,53794	2,174382	0,51897
ADR5_ADRESCYCLUS.WOONPLKODE	SA_NUMMERAANDUIDING.WPL_A1161	-	-	-	1,509779	0,542299	1,154088	-	2,08837	-
ADR5_ADRESCYCLUS.X_KOORD	SA_NUMMERAANDUIDING.X	0,392977	-	-	1,526445	0,558966	1,170754	-	2,105036	-
ADR5_ADRESCYCLUS.Y_KOORD	SA_NUMMERAANDUIDING.Y	0,392977	-	-	1,526445	0,558966	1,170754	-	2,105036	-
ADR5_STRAAT.DDEINDE	SA_OPENBARERUIMTE.DAT_END	0,637233	0,589807	0,637233	1,46719	0,933646	-	-	-	-
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_BEG	-	0,618089	0,665515	1,495473	0,961929	-	0,487667	-	-
ADR5_STRAAT.STRAAT_PTT	SA_OPENBARERUIMTE.STRNM_PTT	0,694699	0,647273	0,694699	1,524657	0,991113	1,417948	0,516851	-	-
ADR5_STRAAT.STRAATBOCO	SA_OPENBARERUIMTE.STRNM_BOCO	0,701747	0,654321	0,701747	1,531705	0,998161	1,424996	0,523899	-	-
ADR5_STRAAT.STRAATKODE	SA_OPENBARERUIMTE.TY_A1117	0,760844	0,713418	0,760844	-	1,057257	-	0,582996	-	1
ADR5_STRAAT.STRAATKODE	SA_OPENBARERUIMTE.STR_CODE	-	-	-	1,530459	-	1,423751	-	-	-
ADR5_WOONPLAATS.DDEINDE	SA_WOONPLAATS.DAT_END	0,546581	0,546581	0,546581	1,567735	0,915331	1,425908	-	1,737927	-
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_BEG	0,574864	-	-	-	-	-	-	-	-
ADR5_ADRESCYCLUS.ADRESNR	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,448725	-	0,415528	1,582194	0,614715	1,226503	-	2,160785	-
ADR5_ADRESCYCLUS.DDEINDE	SA_WOONPLAATS.DAT_END	-	-	-	-	0,69984	-	-	-	-
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_END	-	-	-	1,670932	-	1,315241	-	2,249523	-
ADR5_ADRESCYCLUS.DDINGANG	SA_WOONPLAATS.DAT_END	-	-	-	-	0,728123	-	-	-	-
ADR5_STRAAT.DDEINDE	SA_NUMMERAANDUIDING.DAT_END	-	-	-	-	-	-	-	1,753233	-
ADR5_STRAAT.DDEINDE	SA_WOONPLAATS.DAT_END	-	-	-	-	-	1,383684	-	-	-
ADR5_STRAAT.DDINGANG	SA_NUMMERAANDUIDING.DAT_END	-	-	-	-	-	-	-	1,781516	-
ADR5_STRAAT.DDINGANG	SA_WOONPLAATS.DAT_END	-	-	-	-	-	1,411967	-	-	-
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_END	0,665515	-	-	-	-	-	-	-	-
ADR5_STRAAT.STRAAT_OFF	SA_OPENBARERUIMTE.STR_CODE	0,675473	0,628047	0,675473	1,505431	0,971887	1,398722	0,497625	-	-
ADR5_STRAAT.STRAAT_OFF	SA_NUMMERAANDUIDING.STR_CODE	-	-	-	-	-	-	-	1,791474	-
ADR5_STRAAT.STRAAT_PTT	SA_NUMMERAANDUIDING.STR_CODE	-	-	-	-	-	-	-	1,773511	-
ADR5_STRAAT.STRAATBOCO	SA_NUMMERAANDUIDING.STR_CODE	-	-	-	-	-	-	-	1,784152	-
ADR5_STRAAT.STRAATKODE	SA_NUMMERAANDUIDING.STR_CODE	-	-	-	-	-	-	-	1,816502	-
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_END	-	0,574864	0,574864	1,596018	0,943614	1,454191	0,461402	1,76621	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_WOONPLAATS.DAT_DOC	-	-	-	1,397828	0,745424	1,256001	-	1,56802	-
ADR5_WOONPLAATS.WOONPLKODE	SA_WOONPLAATS.IND_IN_ONDZK	-	-	-	1,396902	0,744498	1,255075	-	1,567094	-
	Number wrong	3	2	3	6	7	8	2	11	0
	Number missed	3	5	5	3	4	5	10	8	16
	Number missed & wrong	6	7	8	9	11	13	12	19	16
	Number right	19	17	17	19	18	17	12	14	6
	missed + wrong : all	**0,24**	**0,291667**	**0,32**	**0,321429**	**0,37931**	**0,433333**	**0,5**	**0,575758**	**0,727273**

Figure 44: Variation test for threshold values

5.8.3 WEIGHTS

Scorers and Algorithms can be given weights in the configuration file. This is done to give a level of importance to a scorer or an algorithm. For example if one algorithm produces good results, but the returned score is just a bit lower than the score from another algorithm, then the weights can be tweaked to give the algorithm just a little more of an edge.

In Figure 45 the results are depicted when the weights of the String Match Algorithms are tweaked. The table has a header with a code where a 1 stands for a weight of 2 and 0 stands for a weight of 1; with from left to right:

- Jaro-Winkler
- Levenshtein
- SoundEx

So for example **101** means a weight of 2 for Jaro-Winkler, a weight of 1 for Levenshtein and a weight of 2 for SoundEx. The distribution of the scores is calculated as follows:

$$multiplier(i) = \frac{weights(i)}{\sum_{j} j \in weights} \cdot \langle \#k : k \in weights \rangle$$

So for the example we get the following distribution of multipliers:

- Jaro-Winkler: 1.2
- Levenshtein: 0.6
- SoundEx: 1.2

In Figure 45 the results are depicted with the various configurations of weights for the String Match algorithms. Again we can see better results for some configurations. However the more correct results, the more faulty results and if there are less faulty results, then there are correct results missing.
The best choice here is to choose the one in the middle, where the weights are equal (**111** or **000**).

		001	101	100	111	000	010	011	110
ADR5_ADRESCYCLUS.POSTK_N	SA_NUMMERAANDUIDING.PC	0,672347	0,677922	0,623995	0,622298	0,622298	0,599006	0,606712	0,610714
ADR5_ADRESCYCLUS.AANC	SA_NUMMERAANDUIDING.AANC	0,680217	0,684959	0,632791	0,632791	0,632791	0,613821	0,618564	0,623306
ADR5_ADRESCYCLUS.DDEINDE	SA_NUMMERAANDUIDING.DAT_END	0,587508	0,599126	0,5255	0,50918	0,50918	0,442988	0,481897	0,474973
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_BEG	-	-	-	-	-	-	-	0,508912
ADR5_ADRESCYCLUS.HUISLT	SA_NUMMERAANDUIDING.HUISLTR_A1130	0,579522	0,590515	0,514736	0,498532	0,498532	0,429793	0,471342	0,462195
ADR5_ADRESCYCLUS.HUISNR	SA_NUMMERAANDUIDING.HUISNR_A1120	0,632995	0,642737	0,580014	0,569828	0,569828	0,524932	0,547453	0,547751
ADR5_ADRESCYCLUS.POSTK_A	SA_NUMMERAANDUIDING.PC	0,672347	0,677922	0,623995	0,622298	0,622298	0,599006	0,606712	0,610714
ADR5_ADRESCYCLUS.STRAATKODE	SA_NUMMERAANDUIDING.STR_CODE	0,63496	0,643318	0,58074	0,572449	0,572449	0,530103	0,551589	0,550896
ADR5_ADRESCYCLUS.TOEV	SA_NUMMERAANDUIDING.TOEV_A1140	0,582995	0,59607	0,528625	0,512421	0,512421	0,454099	0,48523	0,484417
ADR5_ADRESCYCLUS.VOLGNR	SA_NUMMERAANDUIDING.VOLGNR	0,680217	0,684959	0,632791	0,632791	0,632791	0,613821	0,618564	0,623306
ADR5_ADRESCYCLUS.VRY_VELD2	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,680217	0,684959	0,632791	0,632791	0,632791	0,613821	0,618564	0,623306
ADR5_ADRESCYCLUS.WOONPLKODE	SA_NUMMERAANDUIDING.WPL_A1161	0,480911	0,470515	-	-	-	-	-	-
ADR5_ADRESCYCLUS.X_KOORD	SA_NUMMERAANDUIDING.X	0,455911	0,480515	0,42168	0,392977	0,392977	-	-	-
ADR5_ADRESCYCLUS.Y_KOORD	SA_NUMMERAANDUIDING.Y	0,455911	0,480515	0,42168	0,392977	0,392977	-	-	-
ADR5_STRAAT.DDEINDE	SA_OPENBARERUIMTE.DAT_END	0,691849	0,71058	0,653552	0,637233	0,637233	0,59001	0,624177	0,61251
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_BEG	0,713061	0,727474	0,67467	0,665515	0,665515	0,632529	0,658192	-
ADR5_STRAAT.STRAAT_PTT	SA_OPENBARERUIMTE.STRNM_PTT	0,734949	0,747809	0,700089	0,694699	0,694699	0,672774	0,690388	0,681471
ADR5_STRAAT.STRAATBOCO	SA_OPENBARERUIMTE.STRNM_BOCO	0,740235	0,754069	0,707913	0,701747	0,701747	0,680807	0,696814	0,689928
ADR5_STRAAT.STRAATKODE	SA_OPENBARERUIMTE.TY_A1117	0,784557	0,796413	0,760844	0,760844	0,760844	0,760844	0,760844	0,760844
ADR5_WOONPLAATS.DDEINDE	SA_WOONPLAATS.DAT_END	0,634215	0,64109	0,562901	0,546581	0,546581	0,499359	0,533526	0,521859
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_BEG	-	-	-	-	-	-	-	0,555798
ADR5_ADRESCYCLUS.ADRESNR	SA_NUMMERAANDUIDING.HOEK_HUISNR	0,500501	0,51408	0,460859	0,448725	0,448725	0,413272	0,424791	0,435761
ADR5_ADRESCYCLUS.DDINGANG	SA_NUMMERAANDUIDING.DAT_END	0,608721	0,61602	0,546617	0,537463	0,537463	0,485507	0,515912	-
ADR5_STRAAT.DDINGANG	SA_OPENBARERUIMTE.DAT_END	-	-	-	-	-	-	-	0,64645
ADR5_STRAAT.STRAAT_OFF	SA_OPENBARERUIMTE.STR_CODE	0,720529	0,735636	0,684872	0,675473	0,675473	0,644733	0,667955	0,658399
ADR5_WOONPLAATS.DDINGANG	SA_WOONPLAATS.DAT_END	0,655427	0,657984	0,584018	0,574864	0,574864	0,541878	0,567541	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_OPENBARERUIMTE.WPL_A1115	0,446855	0,446332	-	-	-	-	-	-
ADR5_WOONPLAATS.WOONPLBOCO	SA_OPENBARERUIMTE.STRNM_BOCO	-	-	0,357272	-	-	-	-	-
ADR5_WOONPLAATS.WOONPLKODE	SA_OPENBARERUIMTE.WPL_A1115	0,446855	0,446332	-	-	-	-	-	-
	Number wrong	6	6	5	4	4	4	4	3
	Number missed	2	2	3	3	3	5	5	4
	Number missed & wrong	8	8	8	7	7	9	9	7
	Number right	6	6	5	4	4	4	4	3
	missed + wrong : all	**0,571429**	**0,571429**	**0,615385**	**0,636364**	**0,636364**	**0,692308**	**0,692308**	**0,7**

Figure 45: Variation test for String Match algorithm-weights

5.9 DISCUSSION OF RESULTS

The results generated are already pretty good even though not all suggested scorers were implemented in the prototype. For example value matching as mentioned in 2.3.2.2 and type matching (3.3.1.4) are not yet implemented.

Section 5.8 clearly shows that adding more scorers and algorithms leads to better weaving models. Obtained results are always dependent on how the parameters are set in the configuration; therefore this should not be neglected in the future. Varying these parameters is an essential part in generating good results.

The generated weaving model already would save a user (in this case an URBIDATA consultant) a lot of time in setting up an integration process. Using the user's feedback, giving visual interpretations as can be seen in Figure 28 and updating the history will improve the score results. In short preliminary results are satisfactory and practice will reveal on which fronts even more improvement is possible.

6 CONCLUSION AND FUTURE WORK

This chapter provides a description of the results of this project. In section 1, it is evaluated whether the goals of this project were reached. Section 2 provides a recommendation of future work to be done, expanding the achievement of this project. Finally, in section 3, the project itself is evaluated.

6.1 CONCLUSION

The goal of this project was to research whether it is possible to automate the schema matching process in a data integration process. This research is part of the MDM redesign project at URBIDATA, in particular it was part of a research project concerning automated generation of transformation models in a data integration process. In this document, the design of a component, named WMgen, which enables automation of weaving model generation, is described. This automatic generation of weaving models is based on analysis of the source model and comparison to the outcome of previous integration projects. The design is made extendible; new matching algorithms and new score generators can be added easily.

In section 1.6, it was mentioned that in order to automate the generation of weaving models, some analysis of human reasoning in the construction of weaving models and research into whether it is possible to automate this reasoning process was necessary. The reasoning process was divided into two parts: a structural analysis of the data and a semantic analysis of the data.

It was however not possible, within the given time span of the project, to automate the creation of transformation models on the basis of automatically generated weaving models. This means that the WMgen component is a prototype that can be used in further research in order to automate creation of transformation models. More on this can be found in section 6.2

6.2 FUTURE WORK

As was mentioned in the previous section, the automatic generation of transformations is something that still needs to be researched. The difficulty here is how we can automatically find out what a transformation from one object to another looks like. For instance suppose we have hectares as data in the source model and we need square meters in the target model. How can we analyze these sorts of mappings such that we can automatically create the correct transformation?

Furthermore, the time span of this project did not allow for a complete implementation of the redesign project. This project delivered the research that was necessary in order to automate the business process of creating weaving models. The implementation in the redesign project is something that still needs to be done; this research can serve as a basis for the implementation of the weaving model generation component in the redesign project.

The automatic generation of weaving models itself can be expanded as follows; semantic scoring can be improved by automatically adding more metadata to the history and using this metadata in the matching process. If for instance we automatically add something like a description of the transformation to the history and we use this description in the matching process, we can more accurately rate the possibility that a current connection between a source and a

target attribute is the connection that we seek based on the description in the history. We currently only use the weight that is stored in the history for a certain connection between attributes. Also the structure analysis is very dependent on the fact that meaningful names were used for models, objects and attributes. This dependency could be lessened by also using the actual values of attributes in the matching process. Also URBIDATA would like to be able to also automatically generate weaving models of data models that contain geometric data. This was beyond the scope of this project.

Finally, some extra functionalities that will be necessary or at least very useful in the future are the support of different languages, language recognition, a user interface for supplying the system with feedback on the generated weaving model and an extension of MDM by using the FME [19] standard transformation library in generating transformation models. Safe software currently defines 273 transformation operators in 12 categories, which can be used to define a transformation knowledge base.

6.3 PROJECT EVALUATION

In this last section I would like to evaluate this research project. I will start by describing the experience of doing a research project within a company and what I have learned from this experience. Finally I will elaborate on some technical issues that I have researched, used and learned to handle.

Performing a research project for a company often starts out with conflicting goals. The company would like to see results as soon as possible, but research does not give any usable results for the company until the end of the project is reached. Also the company tends to have very high expectations in absence of a deep understanding of the difficulty of the problem that needs to be resolved. Since at the start of this project I also suffered of a lack of deep understanding of the difficulty of the problem, it was for me to find out that the project as it was laid out in the beginning was too large. I had to cut out parts of the initial problem until only a piece of the original problem was left for which it was possible to come up with a solution within the time span of the project. This was done in deliberation with the supervisor at URBIDATA and the supervisor at TU/e. Concluding I can state that this was a valuable experience for me in how a Computer Science graduate is expected to operate in the environment of a company.

As for the technical issues; I have learned a lot about data integration and the processes necessary to integrate large data models, I have learned how to model transformations between large data models and I have had some experience with the standards used in the geo-industry. Furthermore I have had some more practical experience with UML modeling and Java programming and I have gained basic knowledge of semantic web techniques.

But the most valuable, and also the most time consuming, experience was scientific writing. In my experience it was very difficult to state in writing exactly what you mean to say. Some sentences in this document were changed over and over again until they were disambiguated and completely clear to the reader. This is an experience that will be valuable throughout the rest of my career.

REFERENCES

[1] W. E. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau, Washington, DC, 1999.

[2] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Doklady Akademii Nauk SSSR, 163(4):845--848, 1965.

[3] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Valduriez, P. Applying Generic Model Management to Data Mapping. In proc. of BDA 2005, Saint-Malo, France, pp 343—355.

[4] Del Fabro, M. D. and Valduriez, P. 2007. Semi-automatic model integration using matching transformations and weaving models. In Proceedings of the 2007 ACM Symposium on Applied Computing (Seoul, Korea, March 11 - 15, 2007). SAC '07. ACM, New York, NY, 963-970.

[5] David Aumueller , Hong-Hai Do , Sabine Massmann , Erhard Rahm, Schema and ontology matching with COMA++, Proceedings of the 2005 ACM SIGMOD international conference on Management of data, June 14-16, 2005, Baltimore, Maryland

[6] Fellbaum, C. WordNet, an Electronic Lexical Database. MIT Press, 1998. Reference site: http://wordnet.princeton.edu/

[7] The Semantic Web, Tim Berners-Lee, James Hendler and Ora Lassila Scientific American may 17, 2001

[8] Inmon, W. H., Imhoff, C., and Sousa, R. 2000 Corporate Information Factory. 2nd. John Wiley & Sons, Inc.

[9] Paradigms for machine learning, Jaime G. Carbonell, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

[10] Steve Pepper. The TAO of Topic Maps http://www.ontopia.net/topicmaps/materials/tao.html

[11] ISO 13250: Topic Maps. http://www.isotopicmaps.org.

[12] ISO 19115: Geographic Information – Metadata. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26020

[13] Ever wondered how Soundex works?, Patrick Sinke. http://technology.amis.nl/blog/?p=1079

[14] Staff 2006. The Art of Computer Programming, by D.E. Knuth. *Sci. Program.* 14, 3,4 (Dec. 2006), 267-268.

[15] Sesame Framework, Aduna. http://www.openrdf.org/

[16] Chappell, D. 2004 *Enterprise Service Bus*. O'Reilly Media, Inc.

[17] Kimball, R., Reeves, L., Thornthwaite, W., Ross, M., and Thornwaite, W. 1998 *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses with CD Rom*. 1st. John Wiley & Sons, Inc.

[18] URBIDATA bv, 13 december 2006, *Datamodel Gemeentebreed GIS - Gemeente Westland*, Definitief Versie 2.4

[19] Safe Software, FME Translator/Converter, http://www.safe.com/index.php

[20] NEN 3610:2005 nl 'Basismodel Geo-informatie, 2005, http://www2.nen.nl/nen/servlet/dispatcher.Dispatcher?id=253335

[21] Referentiemodel voor het Stelsel van Gemeentelijke Basisgegevens, 2008, http://www.egem-iteams.nl/rsgb

[22] Basisregistratie voor Adressen en Gebouwen, 2008, http://www.egem-iteams.nl/stuf

[23] ArcGIS S-57 Electronical Navigation Chart Nautical Data Model, 2004, http://www.esri.com/news/arcnews/summer04articles/nautical-data-model.html

APPENDIX A – JavaDoc

This appendix shows a summary of the Java Documentation generated for the development of the prototype. It highlights the most important functions and gives a more detailed view of the architecture described in chapter 4.

DATAMODEL

DataModel type class; this is the main class for a DataModel representation. It has a Name and consists of DataModelObjects.

Attributes

Attribute	Notes	Constraints and tags
dataModelObjectList Set<DataModelObject> *Private*	A DataModel consists of Objects. For example in a relational data model this could be a list of tables.	*Default*:
name DataModelName *Private*	The Name of the DataModel.	*Default*:

Operations

Method	Notes	Parameters
addObject() void *Public*	Adds a new DataModelObject named Name to the ObjectSet.	**DataModelObject** dataModelObject [in]
DataModel() void *Public*	Constructor initializing the DataModel class.	**string** name [in]

DATAMODELOBJECT

DataModelObject type class. It has a Name and consists of DataModelAttributes.

Attributes

Attribute	Notes	Constraints and tags
dataModelAttributeList Set<DataModelAttribute> *Private*	A DataModelObject consists of Attributes. For example in a relational data model this could be a field in a table.	*Default*:
name DataModelName *Private*	The Name of the DataModelObject.	*Default*:

Operations

Method	Notes	Parameters
addAttribute() void *Public*	Adds a new DataModelAttribute named Name with type Type to the AttributeSet.	**DataModelAttribute** dataModelAttribute [in]
DataModelObject() void *Public*	Constructor initializing the DataModelObject class.	**string** name [in]

DATAMODELATTRIBUTE

DataModelAttribute type class. It has a Name and Type.

Attributes

Attribute	Notes	Constraints and tags
name DataModelName *Private*	The Name of the DataModelAttribute.	*Default*:
type string *Private*	The Type of the DataModelAttribute. The type is represented as a string to offer an simple expansion method for adding new types. This way type names can be compared without having to normalize types to basic types. Type support can now be handled by the various scorers and algorithms.	*Default*:

Operations

Method	Notes	Parameters
DataModelAttribute() void *Public*	Constructor initializing the DataModelAttribute class.	**string** name [in] **string** type [in]

DATAMODELNAME

DataModelName type class. It consists of the value for a DataModel, DataModelObject or DataModelAttribute Name and a list of synonyms for that value.

Attributes

Attribute	Notes	Constraints and tags
originalValue string *Private*	The originalValue with which the DataModelName object was created. This is usefull for future reference when the value attribute is changed.	*Default*:
synonymSet Set<String> *Private*	A set of synonyms for the name Value. This set is used for creating better scores. For example the strings Street and Road are more similar than an algorithm is able to see.	*Default*:
value string *Private*	The original actual name Value for a DataModel, DataModelObject or DataModelAttribute.	*Default*:

Operations

Method	Notes	Parameters
createSynonymSet() void *Private*	Fixes underscore adaptations, translates abreviations and executes the thesaurus to create a synonymset of the value.	
DataModelName() void *Public*	Constructor initializing the Name class. The Name class generates a list of synonyms using a thesaurus.	**string** value [in]

ABSTRACTRETURNMODEL

AbstractReturnModel represents a class type for WeavingModel and ScoreModel objects. These objects are the result of the Scorer objects.

Attributes

Attribute	Notes	Constraints and tags
connectionSet Set<Connection> *Private*	A ReturnModel consists of Connections. For example a DataModelAttribute from a source DataModel and a DataModelAttribute from a target DataModel are connected.	*Default*:

Operations

Method	Notes	Parameters
addConnection() void *Public*	Adds a new Connection between AObject1 and AObject2 with a score of AScore to the ConnectionSet.	**Object** object1 [in] **Object** object2 [in] **float** score [in]
ReturnModel() void *Public*	Constructor.	

WEAVINGMODEL

A WeavingModel is a special kind of ReturnModel which only has connections between two DataModelAttributes.

Operations

Method	Notes	Parameters
addConnection() void *Public*	Adds a new Connection object containing the score of the connection between a SourceDataModelAttribute and a TargetDataModelAttribute to the ConnectionList.	**DataModelAttribute** sourceDataModelAttribute [in] **DataModelAttribute** targetDataModelAttribute [in] **float** score [in]

SCOREMODEL

A ScoreModel is a kind of ReturnModel that represents a model of scored connections between DataModels, DataModelObjects or DataModelAttributes.

CONNECTION

A Connection connects two objects DataModel, DataModelObject or DataModelAttributes to each other. Such a connection has a score, which represents the certainty of the connections correctness. It also can have a TransformationList which defines how the source object should be transformed to fit in the target object.

Attributes

Attribute	Notes	Constraints and tags
score float *Private*	The Score of a Connection.	*Default*:
sourceObject Object *Private*	The source object of the connection. This can be a DataModel, DataModelObject or DataModelAttribute object.	*Default*:
targetObject Object *Private*	The target object of the connection. This can be a DataModel, DataModelObject or DataModelAttribute object.	*Default*:
transformationList List<String> *Private*	A list of transformations that describe how the source object should be transformed to fit the target object. Currently this is a simple StringList. In the future transformations might be represented as actual objects.	*Default*:

Operations

Method	Notes	Parameters
Connection() void *Public*	Constructor initializing the Connection class. It represents the score of a connection between two objects.	**Object** sourceAbstractDataModel [in] **Object** targetAbstractDataModel [in] **float** score [in]
getTransformationList() List<String> *Public*	Returns the TransformationList.	
setTransformationList() void *Public*	Sets the TransformationList	**List<String>** transformationList [in]

MAIN

The Main method of the WMgen prototype.
The input is a filename containing an XML description of the source and target datamodels. These models are then parsed by XMLToClass and passed ScorerInitiator. The ScorerInitiator results in a set of ReturnModels, which are converted to a WeavingModel bij the ReturnModelComparer. This WeavingModel is passed to ClassToXML and converted back to a WeavingModel XML format.

Operations

Method	Notes	Parameters
Main() string *Public*	The Main method/constructor.	**string** fileName [in]

SCORERINITIATOR

The ScoreInitiator runs all scorers in the order of their priorities. Multiple scorers of the same priorities are started in threads.

Attributes

Attribute	Notes	Constraints and tags
dataModelList1 Set<DataModel> *Private*	The set of source DataModels.	*Default*:
dataModelList2 Set<DataModel> *Private*	The set of target DataModels.	*Default*:
returnModelList Set<AbstractReturnModel> *Private*	The set of ReturnModels calculated by the scorers invoked by the scoreAll method.	*Default*:

Operations

Method	Notes	Parameters
scoreAll() void *Public*	Initiates all scorers and gets the returned ReturnModels.	
ScoreInitiator() void *Public*	Constructor.	**Set<DataModel>** dataModelList1 [in] **Set<DataModel>** dataModelList2 [in]

INTERFACESCORER

InterfaceScorer is an interface for a scorer object. A scorer object determines the score for a comparison between two DataModel objects. Each scorer is configured by the config file, that defines which algorithms should be used. Also each scorer has a priority. For example the StringMatchScorer should be executed before the StructureScorer, that uses the StringMatchScorer's results.

Attributes

Attribute	Notes	Constraints and tags
dataModel1 DataModel *Private*	The source DataModel.	*Default*:
dataModel2 DataModel *Private*	The target DataModel.	*Default*:
priority int *Private*	The priority of the scorer.	*Default*:

Operations

Method	Notes	Parameters
score() AbstractReturnModel *Public*	Compares the source and target DataModel and returns a ReturnModel i.e. a ScoreModel or a WeavingModel.	**DataModel** dataModel1 [in] **DataModel** dataModel2 [in]

STRINGMATCHSCORER

The StringMatchScorer compares DataModelAttribute names and types.

StructureScorer

The StructureScorer calculates the score of an DataModelObject compared to DataModelAttribute scores.

SemanticScorer

The SemanticScorer compares the DataModels to the history of previously constructed WeavingModels. It returns a scored WeavingModel.

InterfaceAlgorithm

InterfaceAlgorithm is an interface for an algorithm object. Each algorithm belongs to a scorer. An algorithm object determines the score for a comparison between two objects. These two objects can be for example a DataModel, a DataModelObject, a DataModelAttribute or even a string.

Operations

Method	Notes	Parameters
score() float *Public*	Abstract score method that returns a score between two objects. These objects can be anything; two strings, two DataModelAttributes or even two objects of different types.	**Object** object1 [in] **Object** object2 [in]

JaroWinklerAlgorithm

JaroWinklerAlgorithm implements AbstractAlgorithm. It calculates the score between two DataModelName objects by using JaroWinkler on the name and the synonyms.

Operations

Method	Notes	Parameters
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are two DataModelName objects.	**DataModelName** name1 [in] **DataModelName** name2 [in]

LevenshteinAlgorithm

LevenshteinAlgorithm implements AbstractAlgorithm. It calculates the score between two DataModelName objects by using Levenshtein distance on the name and the synonyms.

Operations

Method	Notes	Parameters
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are two DataModelName objects.	**DataModelName** name1 [in] **DataModelName** name2 [in]

SoundExAlgorithm

SoundExAlgorithm implements AbstractAlgorithm. It calculates the score between two DataModelName objects by generating the Dutch SoundEx codes for the name and synonyms and then calculating the JaroWinkler score between each two source and target codes.

Operations

Method	Notes	Parameters
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are two	**DataModelName** name1 [in]

Method	Notes	Parameters
	DataModelName objects.	**DataModelName** name2 [in]

TypeMatcherAlgorithm

TypeMatcherAlgorithm implements AbstractAlgorithm. It calculates the score of a comparison between two DataModelAttribute FType fields.

Operations

Method	Notes	Parameters
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are two strings.	**string** type1 [in] **string** type2 [in]

ObjectStructureMatcherAlgorithm

ObjectStructureMatcherAlgorithm implements AbstractAlgorithm. It calculates the score of a comparison between a source DataModelObject and target DataModelObjects by looking at the ScoreModels for the DataModelAttributes of those objects.

Operations

Method	Notes	Parameters
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are a source DataModelObject and a Scoremodel.	**DataModelObject** dataModelObject [in] **ScoreModel** scoreModel [in]

HistoryMatcherAlgorithm

HistoryMatcherAlgorithm implements AbstractAlgorithm. It calculates the score of a comparison using a history database between two DataModelAttributes and retrieves a Set of Transformation strings from that history database.

Attributes

Attribute	Notes	Constraints and tags
transformationList List<String> *Private*	The transformations retrieved from the History Database	*Default*:

Operations

Method	Notes	Parameters
getTransformationList() List<String> *Public*	Return the TransformationList.	
score() float *Public*	Implements the score method of AbstractAlgorithm. The inputs are two DataModelAttribute objects.	**DataModelAttribute** dataModelAttribute1 [in] **DataModelAttribute** dataModelAttribute2 [in]

ReturnModelComparer

ReturnModelComparer takes a set of ReturnModels and converts them to a WeavingModel by reviewing the best score combinations.

Operations

Method	Notes	Parameters
ReturnModelComparer() WeavingModel	Constructor.	**Set<AbstractReturnModel>** returnModelList

Method	Notes	Parameters
Public		[in]

XMLToClass

Converts the input XML to the DataModel class structure.

Operations

Method	Notes	Parameters
fixPrefixes() void *Private*	Removes unnecessary prefixes from DataModelName values throughout a DataModel object.	
XMLToClass() DataModel *Public*	Constructor.	**string** fileName [in]

ClassToXML

Converts the resulting WeavingModel to XML and outputs it to the system out.

Operations

Method	Notes	Parameters
ClassToXML() void *Public*	Constructor.	**WeavingModel** weavingModel [in]

APPENDIX B – XML FOR INPUT DATAMODELS

This appendix contains the XML schema which describes the input data model feed for the WMgen prototype.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

	<xsd:simpleType name="KindType">
		<xsd:restriction base="xsd:string">
			<xsd:enumeration value="source"/>
			<xsd:enumeration value="target"/>
		</xsd:restriction>
	</xsd:simpleType>

	<xsd:element name="DATAMODELS" type="DataModelsType"/>

	<xsd:complexType name="DataModelsType">
		<xsd:sequence>
			<xsd:element name="DATAMODEL" type="DataModelType"/>
		</xsd:sequence>
	</xsd:complexType>

	<xsd:complexType name="DataModelType">
		<xsd:element name="OBJECTS" type="ObjectsType"/>
		<xsd:attribute name="NAME" type="xsd:string"/>
		<xsd:attribute name="KIND" type="KindType"/>
	</xsd:complexType>

	<xsd:complexType name="ObjectsType">
		<xsd:sequence>
			<xsd:element name="OBJECT" type="ObjectType"/>
		</xsd:sequence>
	</xsd:complexType>

	<xsd:complexType name="ObjectType">
		<xsd:element name="ATTRIBUTES" type="AttributesType"/>
		<xsd:attribute name="NAME" type="xsd:string"/>
	</xsd:complexType>

	<xsd:complexType name="AttributesType">
		<xsd:sequence>
			<xsd:element name="ATTRIBUTE" type="AttributeType"/>
		</xsd:sequence>
	</xsd:complexType>

	<xsd:complexType name="AttributeType">
		<xsd:element name="NAME" type="xsd:string"/>
		<xsd:element name="TYPE" type="xsd:string"/>
	</xsd:complexType>

</xsd:schema>
```

The XML schema can be interpreted as depicted in the example below.

```
<?xml version="1.0" encoding="utf-8" ?>
<DATAMODELS>
       <DATAMODEL NAME=”datamodel_1” KIND=”source”>
              <OBJECTS>
              <OBJECT NAME=”object_1”>
                     <ATTRIBUTES>
                     <ATTRIBUTE>
                            <NAME>attribute_1</NAME>
                            <TYPE>type_1</TYPE>
                     </ATTRIBUTE>
                     :
                     :
                     <ATTRIBUTE>
                            <NAME>attribute_n</NAME>
                            <TYPE>type_n</TYPE>
                     </ATTRIBUTE>
                     </ATTRIBUTES>
              </OBJECT>
              :
              :
              </OBJECTS>
       </DATAMODEL>
       <DATAMODEL NAME=”datamodel_2” KIND=”target”>
       :
       :
       </DATAMODEL>
</DATAMODELS>
```

APPENDIX C – XML FOR OUTPUT WEAVINGMODELS

This appendix contains the XML schema which describes the weaving model for the output of the WMgen prototype. The XML output can then be converted to a simple transformation model.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

	<xsd:simpleType name="KindType">
		<xsd:restriction base="xsd:string">
			<xsd:enumeration value="source"/>
			<xsd:enumeration value="target"/>
		</xsd:restriction>
	</xsd:simpleType>

	<xsd:element name="WEAVINGMODEL" type="WeavingModelType"/>

	<xsd:complexType name="WeavingModelType">
		<xsd:element name="CONNECTIONS" type="ConnectionsType"/>
	</xsd:complexType>

	<xsd:complexType name="ConnectionsType">
		<xsd:sequence>
			<xsd:element name="CONNECTION" type="ConnectionType"/>
		</xsd:sequence>
	</xsd:complexType>

	<xsd:complexType name="ConnectionType">
		<xsd:element name="SOURCEATTRIBUTES" type="AttributesType"/>
		<xsd:element name="TARGETATTRIBUTES" type="AttributesType"/>
		<xsd:element name="TRANSFORMATIONS"
type="TransformationsType"/>
		<xsd:element name="SCORE" type="xsd:float"/>
		<xsd:element name="VERIFIED" type="xsd:boolean"/>
	</xsd:complexType>

	<xsd:complexType name="AttributesType">
		<xsd:sequence>
			<xsd:element name="ATTRIBUTE" type="AttributeType"/>
		</xsd:sequence>
		<xsd:attribute name="KIND" type="KindType"/>
	</xsd:complexType>

	<xsd:complexType name="AttributeType">
		<xsd:sequence>
			<xsd:element name="DATAMODEL" type="xsd:anyURI"/>
			<xsd:element name="OBJECT" type="xsd:anyURI"/>
			<xsd:element name="ATTRIBUTE" type="xsd:anyURI"/>
		</xsd:sequence>
	</xsd:complexType>

	<xsd:complexType name="TransformationsType">
		<xsd:sequence>
			<xsd:element name="TRANSFORMATION" type="xsd:string"/>
		</xsd:sequence>
	</xsd:complexType>

</xsd:schema>
```

The XML schema can be interpreted as depicted in the example below.

```
<?xml version="1.0" encoding="utf-8" ?>
<WEAVINGMODEL>
       <CONNECTIONS>
       <CONNECTION>
              <SOURCEATTRIBUTES>
              <SOURCEATTRIBUTE>
                     <DATAMODEL>datamodel_1</DATAMODEL>
                     <OBJECT>object_1</OBJECT>
                     <ATTRIBUTE>attribute_1</ATTRIBUTE>
              </SOURCEATTRIBUTE>
              :
              :
              <SOURCEATTRIBUTE>
                     <DATAMODEL>datamodel_n</DATAMODEL>
                     <OBJECT>object_n</OBJECT>
                     <ATTRIBUTE>attribute_n</ATTRIBUTE>
              </SOURCEATTRIBUTE>
              </SOURCEATTRIBUTES>
              <TARGETATTRIBUTES>
              <TARGETATTRIBUTE>
                     <DATAMODEL>datamodel_1</DATAMODEL>
                     <OBJECT>object_1</OBJECT>
                     <ATTRIBUTE>attribute_1</ATTRIBUTE>
              </TARGETATTRIBUTE>
              :
              :
              <TARGETATTRIBUTE>
                     <DATAMODEL>datamodel_n</DATAMODEL>
                     <OBJECT>object_n</OBJECT>
                     <ATTRIBUTE>attribute_n</ATTRIBUTE>
              </TARGETATTRIBUTE>
              </TARGETATTRIBUTES>
              <TRANSFORMATIONS>
              <TRANSFORMATION>transformation_1</TRANSFORMATION>
              :
              :
              <TRANSFORMATION>transformation_n</TRANSFORMATION>
              </TRANSFORMATIONS>
              <SCORE>1</SCORE>
       </CONNECTION>
       :
       :
       <CONNECTION>
              <SOURCEATTRIBUTES>. . . .</SOURCEATTRIBUTES>
              <TARGETATTRIBUTES>. . . .</TARGETATTRIBUTES>
              <TRANSFORMATIONS>. . . .</TRANSFORMATIONS>
              <SCORE>1</SCORE>
       </CONNECTION>
       </CONNECTIONS>
</WEAVINGMODEL>
```

APPENDIX D – XML FOR INPUT WEAVINGMODELS

This appendix contains an XML example for feedback weaving models as described in section 2.3.2.1. The XML basically follows the same schema as described in appendix C with a difference that the *VERIFIED*-tag is added. Such an input weaving model enables users to update the history knowledge base of WMgen. *VERIFIED* indicates whether the user thought the transformation rule was correct and *SCORE* indicates the certainty percentage of the user. Also a *CONNECTION* can be changed by a user, this enables the user to add new information or change old information in the history knowledge base.

```
<?xml version="1.0" encoding="utf-8" ?>
<WEAVINGMODEL>
       <CONNECTIONS>
       <CONNECTION>
              <SOURCEATTRIBUTES>. . . .</SOURCEATTRIBUTES>
              <TARGETATTRIBUTES>. . . .</TARGETATTRIBUTES>
              <TRANSFORMATIONS>. . . .</TRANSFORMATIONS>
              <SCORE>1</SCORE>
              <VERIFIED>true</VERIFIED>
       </CONNECTION>
       :
       :
       <CONNECTION>
              <SOURCEATTRIBUTES>. . . .</SOURCEATTRIBUTES>
              <TARGETATTRIBUTES>. . . .</TARGETATTRIBUTES>
              <TRANSFORMATIONS>. . . .</TRANSFORMATIONS>
              <SCORE>1</SCORE>
              <VERIFIED>false</VERIFIED>
       </CONNECTION>
       </CONNECTIONS>
</WEAVINGMODEL>
```

APPENDIX E – THE COMPLETE WESTLAND DATAMODEL

This appendix contains the complete model of the Westland project carried out by URBIDATA. A subset of this model was used in the test cases depicted in chapter 5.