Universiteit Leiden
The Netherlands

Opleiding Informatica & Economie

Aligning an industry data model

with applications in real estate

Pepijn Griffioen

Supervisors:
Dr. M.C. van Wezel & Dr. W.A. Kosters & Dr. P.W.H. van der Putten

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2018

Abstract

Recently a sector initiative called VERA (Volkshuisvesting Enterprise Referentie Architectuur) has been defined in the real estate industry. It is a very detailed data model that defines what and how data is formatted within the real estate industry. It enables housing organizations to exchange housing data seamlessly with one another, but in this industry there are many software vendors with their own proprietary model. For these vendors to change their data model to the VERA data model is an expensive operation, and as a solution the creation of an API was proposed. The vendors can create a mapping from their data models to the API, than the API can deliver the vendor data in the VERA format. Such an API does not exist yet, so this research was set up.

This research focuses on defining how to generate an API that is implementable for vendors with other data models, generic, consistent over classes and extensible for data not part of the VERA model. In building this API it is important to enable the vendor to map its model to the VERA model which is in the API, so different mapping problems that can occur are identified and solutions for these problems are proposed. For the API two approaches are defined one with all classes as API endpoints and the other with certain important classes as API endpoints. The research concludes that mapping the vendor model to the VERA model is possible with the proposed solutions and the API can be implemented by the vendors. For future work it is interesting to look into other existing Industry Standard Data Models and what problems vendors face that were not identified in this research.

Contents

Chapter 1

Introduction

Social housing organizations provide housing for households with below average income in the Netherlands. The industry provides housing for approximately 2.5 million households in the Netherlands. In the business process of delivering the housing service, several information technology systems are used. These systems support all sorts of processes from monitoring maintenances to checking for rent. In today's society these systems are key to support business practices.

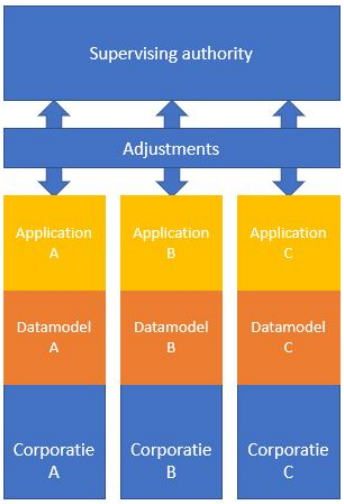
The cost of such information systems is substantial in this industry. Per household it amounts to approximately 175 euros per year. One of the underlying reasons for this high cost is that many vendors use a proprietary data model. Also every social housing organization can have different software vendors that provides information systems to support their business processes. The different vendors have their own proprietary data model, which makes it impossible to easily exchange the data between the corporations.

The complexity of data exchange introduces a vendor lock-in and the need for costly adapters between systems. The vendor lock-in is due to the high costs from changes and adaptations in the data model, when switching to another vendor. The short term solution are adapters between systems, because they do need to be continuously custom built when a change in the data model occurs. The data exchanged between the different information systems needs to be adapted to make sure it meets the needs of the other system. This process is nowadays executed by hand or with adapters, which makes it an expensive operation. Due to the vendor lock-in possible innovations are difficult to create. The vendor lock-in also creates a monopoly environment, where the different vendors have power over their clients, meaning the vendors can ask higher fees for their software than they could in a market of free competition. The adapters are short term solutions and the vendor lock-in is a long term problem that can only be solved collectively.

In Figure 1.1 an abstract overview can be found of how the different housing corporations exchange data with the supervisor of the social housing market. This is just one example. What becomes clear from the example is that the different housing corporations have their own applications and a proprietary data model. When a data exchange needs to happen one needs to make sure the other system can use the data and therefore adjustments need to be made. The adjustments are visualized in the adjustments layer, which is an expensive

operation just for data exchange.

Figure 1.1: Data transfer between parties with the needed adjustments.



To address these problems a sector initiative called VERA (Volkshuisvesting Enterprise Referentie Architectuur) has been defined to achieve standardization in this sector. It is designed for data exchange following the CORA (Corporate Reference Architecture) initiative. CORA defines the reference models, architectural questions, related principles and standards from a business view. VERA is the method from a technical perspective to standardize information exchange in the framework defined by CORA [VER16d] . It has two main goals: reach harmonization in business processes, managing business processes and data exchange within business partnerships, and stimulate software vendors to develop standardized solutions with a better link-ability.

Therefore the integration of the information provision systems is important. IT systems need to work together to support and track changes, as our technologies keep changing at a rapid pace. The goal of VERA is to standardize data exchange. So the implementation of the VERA standard will lead to simpler and better manageable integration projects and a better manageable information provision system [VER16d].

Currently the VERA standard comprises a canonical data model with 121 classes and 1789 attributes, and a single API supports one process, the distribution of housing options for citizens (in dutch: woonruimteverdeling). However, without adoption by suppliers, a standard is useless. This adoption is problematic for various reasons, most importantly:

1. The mapping between proprietary and standardized data model cannot be made on an individual class level

2. A clear specification of which APIs to implement is lacking.

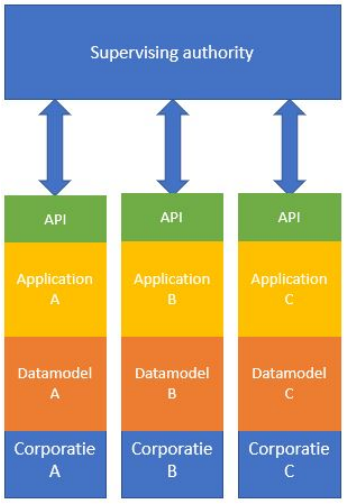
The given adoption problem can hopefully be solved by defining a functional API strategy. The API strategy will define what needs to be implemented to be VERA compliant. From within the industry some API requirements have been created. The functional level API strategy for VERA will be designed such that:

- The API is implementable for vendors, the mapping between data model and the API methods can reasonably be made. (The API must be sufficiently denormalized.)

- The API is generic: the whole object graph can be traversed and manipulated with the API.
- The API is simple: it is consistent over classes, which makes it easier to use for developers.
- The API can be extended for data not part of the VERA model.

Such an API will create the possibility to exchange data between systems and corporations seamlessly. In Figure 1.2 an abstract overview of such a solution is shown. The different applications will be connected to the API, which will convert their data into the VERA standard. Other systems can then retrieve data from the application through the API and because this data is in the VERA standard the other application can integrate it easily in its system. The difference from Figure 1.1 is the removal of the layer of adjustments; this layer is integrated into the API layer, therefore automating the adjustment process. By using the API the different systems become VERA compliant, enabling the exchange of information without expensive adjustments.

Figure 1.2: Data transfer between parties, API that takes care of the adjustments.



The problem has been known for quite some time, but has not been solved in the past. This is due to the costs of investment and the gain from this investment. The problem is faced by all the different vendors in the industry, but investing as a single vendor in such a solution is expensive and it creates no direct gain in revenue. Therefore most vendors have not implemented the standard yet. This investment should come from a collective of vendors rather than a single vendor. To address this, research has been set up to create a functional API strategy for VERA. The research will look into best practices in defining an API and how another data model can be mapped to the API. The main question in this research is as follows:

> How to generate an API specification based on the VERA class model such that the API is implementable for vendors with other data models, generic, consistent over classes and extensible for data not part of the VERA model?

Besides this main question two subquestions will be discussed, which is ”how to design an API?” and ”what problems occur when mapping a vendor model to the VERA model?”. With this main question and the two subquestions a functional API strategy can be defined which makes it easier for the different software vendors to implement the VERA standard and become VERA compliant: to align an industry with an industry data

model in the real estate industry.

1.1 Thesis overview

The remainder of this thesis is structured as follows: Chapter 2 includes related work; Chapter 3 discusses the analysis and methods used to answer the different questions in Chapter 1; Chapter 4 proposes the solutions and evaluates the API; Chapter 5 discusses the research; Chapter 6 concludes.

This bachelor thesis is the final assignment for the bachelor study Computer Science and Economics at the Leiden Institute of Advanced Computer Science from Leiden University. It is supervised by M.C. van Wezel, W.A. Kosters and P.W.H. van der Putten.

Chapter 2

Related Work

In the first chapter we discussed the problem within the market and what questions can be examined to solve this problem. This chapter will look into existing industry standard data models and what normalization and denormalization can mean for the VERA standard.

2.1 Industry standard data models

The VERA model is a standard model for the real estate industry. In the landscape of data models there are more standard data models or industry standard data models. The standard data model or industry standard data model (ISDM) is a model that is defined within a certain industry or domain to standardize data structure, enabling information sharing without modifications.

The ISDM's occur in all sorts of sectors. Some examples of ISDMs are:

1. School Interoperability Framework (SIF)
2. Pipeline Open Data Standard (PODS)

The School interoperability framework (SIF) is an initiative developed in Australia, the United States and the United Kingdom [Ass10] to ensure that educational applications work together more effectively. Through facilitating data sharing between applications, enhance product functionality and providing best practice solutions to customers. This is reached through defining a platform where applications from different vendors can exchange information with one another.
The Pipeline Open Data Standard (PODS) is an initiative that defines the Pipeline Data Model used to store information and analysis data about pipeline systems. It is an industry standard with the aim to eliminate data redundancy and define a single inter-operable data model [Ass].
The two mentioned ISDMs are industry specific, the European Union is also embracing standard data models, but does it in another way. It has defined the European Interoperability Framework to define ISDMs.

The European Interoperability Framework (EIF) is a framework [fIEC17] that is not an ISDM, but it does give guidelines to define ISDMs. It is a conceptual model with mostly recommendations on keeping public digital services inter-operable within the European Union. The framework gives guidelines to define National Interoperability Frameworks (NIFss) or Domain Interoperability Frameworks (DIFs), which are not ISDMs but standard models for their level of information governance [fIEC17].

2.1.1 The VERA model and the ISDMs

SIF and VERA have in common that they both have the aim to define a standard for information exchange. The SIF model in its first form exists since 2000, so the development of SIF is on another level than the VERA standard. Therefore implementations of SIF with proprietary vendor models do exist and might be interesting to look into.

PODS and VERA both have clear data models that are divided into different domains within their industries. The standards have the same aim to define an industry standard to enable efficient information exchange. Also PODS has been around since 1998, and has several possible ways to be implemented from a vendor that has its whole platform built on the data model to vendor implementations that connect to their systems.

Both ISDMs are established data models and could be looked into, but it is a broad and whole other field of research that is interesting to look into in another research project. However, the established ISDMs can be researched to look for best practices that are of importance for the VERA model.

2.2 Normalization and denormalization

In the process of class design a level of normalization can be used. The level of normalization depends on the required performance of the class and the need to divide the different classes further or not. In the standard model the normalization and denormalization principles have also been used to design the model. Next the two terms will be explained.

Normalization is the process of transforming a schema into normal form, with the primary goal of elimination of data redundancy [KL18]. When a schema is in such form it has certain properties such as no repeating groups, atomic values and no transitive dependencies [Ram03]. Two examples of normal forms are Boyce Codd Normal Form (BCNF) and Third Normal Form (3NF). As normalization is mainly focused on database design its influence can also be found in class diagrams.

Denormalization is the process of reconsidering schema decompositions carried out for normalization. With the aim to improve the performance of queries that involve many attributes from several relations [Ram03]. The disadvantage of denormalization is that it implies data loss, so reconstructing normalized data from denormalized data may not be possible. This needs to be taken into account when considering denormalization.

The data model of VERA is a very normalized schema, since most of the data stored within the model has its own class. That makes it almost impossible to have data redundancy within the whole model, but this makes it also very difficult to create a mapping to it. Many vendors that will adapt the VERA model do not have such detailed and normalized data models, therefore it might be interesting to look into the possibility to use denormalization on the VERA model to create a model that a vendor can map its data model to.

2.3 Design patterns

Design patterns are patterns that can be used in designing software solutions. Gamma et al. introduced design patterns as: ”descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” [GHJV94]. There are many defined design patterns, all solving some sort of design problems. For example the adapter pattern that enables two classes to work together that would not be able without the pattern, but there are many other design patterns. Such patterns can also be used when implementing the VERA API. That way certain vendor classes can be modified to fit into the VERA API, re-usability of design patterns to format the data for the API can be reached.

For this research an API was chosen as an option to enable vendors to serve the VERA format, but in the paper of Lano and Kolahdouz-Rahimi UML-RSDS is used [LKR11a]. UML-RSDS is a model-driven development method, that is being used for transformation implementations from specifications [LKR11b]. This method needs models such as class diagram models, use cases, object models and more. With all these models defined the patterns from Lano and Kolahdouz-Rahimi can be used to define on an abstract level a model transformation [LKR11a]. This approach is problematic as it requires the vendors to have their models and documentation in these defined models and some vendors do not have their data model defined in such models.

Chapter 3

Analysis

In this chapter a mapping between a vendor and the VERA model will be made in Section 3.1, then the different problems that occur in a mapping process will be discussed in Section 3.2 and finally the process of designing an API will be discussed in the last section.

3.1 Mapping a vendor to VERA

The mapping is made from a vendor model to the VERA model. For this research a software vendor made its data model available to examine a mapping from a vendor model to the VERA model. The data model that was made available is used in one of their applications, which is an Enterprise Resource Planning application for real estate businesses.

The mapping process is analyzed with knowledge about VERA gained through reading the VERA documentation and meetings with several VERA experts. The used documentation on VERA can be found on the website of the VERA foundation [VER]. The documentation on the data model of the vendor consisted of vendors data model in an entity model and in XML format. A generated HTML file with information on the vendor data model was also available for research purposes, but the description on the different classes and attributes was limited. With the available information two classes in VERA were mapped to the vendor model as an example for the VERA standard. The two VERA classes are ”Relatie” and ”Eenheid”. These two were chosen as they are classes with many attributes and are often referenced to in other VERA classes.

The class ”Relatie” is a legal person, natural person or group of persons that in the past, present or future had something to do with a housing corporation [VER16b].

The class ”Eenheid” is a definition for a delineated real estate object or a defined terrain what can be used in the service provision of the housing corporation [VER16c]. It is something the corporation makes agreements upon with its customers. It is subject to administrative purposes for rent and sale, but also more properties which can be found in the description of the real estate domain in [VER16c].

The mapping from the vendor model to the VERA model was done by hand. The VERA classes were taken as the output, where the mapping from the vendor had to go to. The Entity model of the vendor model that was available for the research was then used to identify possible classes and attributes within the vendor model that could then be mapped to the VERA model. This process was done by hand and the available information was for VERA the documentation on all the classes which is very detailed and can be found on their website [VER]. The available information on the vendor data model was the Entity model that was made available for this research. As the vendor information was only an Entity model the mapping was mostly based on the naming of the different classes and attributes. So there might be small mistakes within the mapping, but that is not a problem as the mapping is only used as an example. The categories that are in the different tables in the subsections are defined using common problems that were found when mapping the vendor model to the VERA model.

3.1.1 Vendor to ”Relatie”

The class ”Relatie” in VERA inherits most of its attributes from three other classes: ”NatuurlijkPersoon”, ”Rechtspersoon” and ”Relatiegroep”. In the mapping process these three classes were also examined in their ability to be mapped with the vendor. For the mapping process the VERA classes were fixed and the vendor model was used to find similar classes and attributes. In the mapping process different categories were identified that indicate problems, these problems are further defined in Section 3.2. In Table 3.1 a explanation per category can be found. The column # defines the number of category occurrences in the mapping of a vendor model to the VERA class ”Relatie”, that gives a perspective on the occurrence of a category. In total there are 52 attributes in the ”Relatie” class and its derivatives.

Table 3.1: Explanation per category from the mapping of ”Relatie”.

Category	Meaning	Example	#
VERA-generated	This attribute is generated in the VERA model and not by the vendor.	The attribute ”id” in the VERA class ”Relatie”. Defined in the VERA model and it needs to be correctly mapped from the vendor model.	16
Not found	The attribute name in the VERA model was not found through checking the names of classes and attributes in the vendor model. This does not state that it is not in the vendor model.	The attribute ”auditInfo” in the VERA class ”Relatie”. In the mapping process the attribute was not found in the vendor model, but it could be possible that it exists in the vendor model.	9
Check attributes	This means that an attribute of the vendor class can be mapped to the VERA class, but that there might be a difference in the attributes that are mapped to the VERA class. So it is important to check which attributes can be mapped and what the type is.	The VERA attribute and a vendor attribute are similar, but they cannot be mapped one to one. They need to be verified if they are similar, because there might be differences between the classes.	29
More possible vendor classes	One class was found that could be mapped to the VERA attribute, but there might be more than just this one vendor class suitable for the VERA class. The vendor model needs to be checked for other possible classes that could be mapped.	The VERA attribute ”signaleringen” in the class ”NatuurlijkPersoon” can have multiple attributes in the vendor model that fullfill the requirement of being a warning for the supplier of the housing option. This can be an attribute in the form of a warning, but it can also be a penalty, both can be used in the VERA attribute.	2

The vendor model can be mapped to the VERA class ”Relatie” for most of the attributes. Of course there are still the attributes with the category ”Not found”, ”Check attributes” and ”More possible vendor classes”, but for someone with knowledge of the vendor model, the attributes with the last two categories can be mapped from vendor to VERA. Even the attributes with ”Not found” might be found by someone with enough

knowledge of the vendor model. If the ”Not found” attributes can still not be found, one cannot map this attribute.

3.1.2 Vendor to ”Eenheid”

The class ”Eenheid” is part of the domain vastgoed. An ”Eenheid” can be a house, garage, business premises, student housing or a administrative object. It can be a ”Eenheid” on itself, it can be part of a group of multiple instances of ”Eenheid” or it can have a structure of ”Eenheid” children. The class has the largest number of attributes in the whole VERA domain, which are 95 in total. Most of the vendors do not have such a big entity to describe something similar to ”Eenheid”. So many attributes cannot be mapped from the vendor model to ”Eenheid”. In the mapping process different categories were identified that indicate problems, these problems are further defined in Section 3.2. In Table 3.2 an explanation per category can be found. The column # defines the number of category occurrences in the mapping of a vendor model to the VERA class ”Eenheid”. There are 95 attributes in total in the ”Eenheid” class.

Table 3.2: Explanation per category from the mapping of ”Eenheid”.

Category	Meaning	Example	#
VERA-generated	This attribute is generated in the VERA model and not by the vendor.	The attribute ”id” in the VERA class ”Relatie”. Defined in the VERA model and it needs to be correctly mapped from the vendor model.	4
Combining data	The attribute might be found through combining different attributes from the vendor model, but it is not clear which ones, or if it is possible.	A VERA attribute that can be found through combining attributes is ”kamersAantal”, it defines the number of rooms in a real estate object. In some vendor models such an attribute is not specifically defined, but it could be found when combining attributes in the vendor model.	4
Not found	The attribute name in the VERA model was not found through checking the names of classes and attributes in the vendor model. This does not state that it is not in the vendor model.	The attribute ”auditInfo” in the VERA class ”Relatie”. In the mapping process the attribute was not found in the vendor model, but it could be possible that it exists in the vendor model.	24
Not mapped	The attribute is part of the attributes which are not tried to be mapped to the VERA model, as they are very detailed. The level of documentation on the vendor model was not detailed enough to create a mapping to the attribute.	The VERA attribute bouwnummer, is a very specific attribute that was in the mapping process not tried to be mapped. It is a VERA specific attribute that might exist in the vendor model, but with the available documentation it was not possible to identify it.	45
Check attributes	This means that an attribute of the vendor class can be mapped to the VERA class, but that there might be a difference in the attributes that are mapped to the VERA class. So it is important to check which attributes can be mapped and what the type is.	The VERA attribute and a vendor attribute are similar, but they cannot be mapped one to one. They need to be verified if they are similar, because there might be differences between the classes.	12

Table 3.3: Explanation per category from the mapping of ”Eenheid”, continued.

Category	Meaning	Example	#
Depends on implementation	The structure of this attribute is clear in the VERA model, but not in the vendor model. Therefore it is needed to check how it is implemented in the vendor model when mapping it to the VERA model.	The VERA attribute ”bovenliggendeEenheid” in the class ”Eenheid” is an attribute that defines a specific structure namely an instance of ”Eenheid” that is a parent to the current instance of ”Eenheid”. The mapping depends on how the vendor model is implemented, how the structure of its objects similar to ”Eenheid” are specified.	3

After a series of attributes in VERA that were not able to be mapped from the vendor model to the VERA model with the level of documentation available for this research, the mapping was cancelled. The need for extra information is important when going into detail in the mapping of two data models.

Some examples of VERA attributes that a vendor model might not have are a ”marktlabel” and ”situering”. The attribute ”marktlabel” is a label that is used for strategic storage policy of the real estate objects. The attribute ”situering” defines a description about how the real estate object is situated, for example the object is next to a busy street. It can be the case that a vendor model does not have such specific attributes clearly defined in its model, therefore making it hard to deliver these attribute in the VERA format.

Some of the VERA attributes might be found in the vendor model if the vendor attributes are combined, but such a practice requires a detailed description of what the vendor attributes define. A VERA attribute that could be found through combination of attributes is ”kamersAantal”, that defines the number of rooms in a real estate object. Such an attribute is not specifically defined in the vendor model, but it could be found when combining attributes from classes with information on a housing option from the vendor model.

From the process of mapping the two different models to one another it is clear that knowledge from the vendor and VERA domain is needed to even make these mappings. Also there is not one way to map a vendor model to the VERA model, adjustments from the vendor model are needed to create a mapping. The mapping was built using the information available, so the mapping was defined using the names of the vendor model and the VERA documentation. If someone would recreate a mapping using the available information it will be almost the same, but because of one’s understanding of attribute and class naming there can be differences in the mapping.

3.2 Mapping problems

A mapping from a vendor to VERA cannot be made without knowledge and understanding of the models. The models have different classes and attributes and different definitions of the classes and attributes.

For example a vendor class similar to ”Relatie” cannot be simply mapped to the VERA class ”Relatie”, the classes have somewhat the same attributes and define a similar entity. Still most of the attributes have different names and definitions, and some attributes are included and some are not. This makes it hard to simply map the two classes to one another, even if they have a lot in common.

The goal of this chapter is to specify on an abstract level what problems occur when mapping a vendor model to the VERA model. The vendor model is taken as an example to examine the different problems in mapping from a vendor perspective. From Section 3.1 the different categories are taken as problems, and other problems that occurred in the mapping process. This section will describe the different problems with examples that occur when creating such a mapping. The problems are divided into the following categories:

1. Information only available in the standard model.
2. Information only available in the vendor model.
3. Information available in both models.

Here the standard model stands for the VERA data model. The different examples that define the problem will use the VERA and vendor models as examples. The names of the classes and attributes of the vendor model are changed to ensure confidentiality. First the problems that occur with information available in the standard model will be defined. In Table 3.4 for each category an example of a problem is given.

Table 3.4: Example problem per category.

Category	**Example**
Only standard model	The standard model has an attribute that the vendor model does not have.
Only vendor model	The vendor model has more attributes than the standard model.
Both models	Not all attributes will be filled.

The categories are specified with the different problems that were found in the mapping process. This resulted in three categories that define a clear overview of the different problems. These categories are found through the mapping process which was done by hand, this does not state that all the problems in a mapping process are found. There are more problems, but we did not come across them, in future research more problems might be added to the defined problems.

3.2.1 Information only available in the standard model

The problems that are defined in this section are problems that are caused by some form of information availability only in the standard model.

Problem 1: The standard model has an attribute that the vendor model does not have.
The standard model has defined very specific attributes. Therefore the standard model is not the same as every vendor model: some standard model attributes do not occur in the vendor model.
Example 1: In the domain ”Overeenkomsten” in VERA there is the class ”Huurgeschil”. In this class there is a attribute ”hoorzittingsdatum”. This attribute refers to a date when the hearing of a rental dispute took place or will take place. In the model of the vendor there is not such an attribute to track this date.

Problem 2: The standard model has a class that the vendor model does not have.
How the standard model is defined does not always guarantee that all the different vendors have the classes that are in the standard model and therefore it can occur that the standard model has a class that is not in the vendor model.
Example 2: In VERA there is a class ”Opleiding”, which defines the current study of a certain relation. The class is added to make sure that social housing corporations who deal with students can assure that students use the student specific housing options. In this research the vendor model used does have a class to define the current study of a relation, but there are vendors that do not deal with students. Therefore they do not have a class that defines the study of a relation.

3.2.2 Information only available in the vendor model

The problems that are defined in this section are problems that are caused by some form of information availability only in the vendor model.

Problem 1: The vendor model has more attributes than the standard model.
A class in the vendor model has more attributes than the class in the standard model, making it not possible to put these attributes in the standard model.
Example 1: The vendor class has an attribute that the corresponding class ”Relatie” inherited from ”NatuurlijkPersoon” does not have. This makes it not possible to exchange the attribute through the VERA model.

Problem 2: The vendor model has a class that the standard model does not have.
Due to the way the standard model is defined not all the vendor models are the same as the standard model. Some classes of the vendor model do not exist in the standard model.
Example 2: In the model of the vendor there is a class ”Inspector” that defines a person that is an inspector and has the right to do inspections. In VERA there is a class that defines inspections called ”Inspectie”, but there is no class with an inspector. The ”Inspectie” class does have a list of persons of the type ”Relatie” that are part of this inspection, but VERA does not have a specific inspector class.

3.2.3 Information available in both models

The problems that are defined in this section are problems that are caused even if the information needed to map one model to the other is available.

Problem 1: Not all attributes will be filled.
Not all attributes in the different classes are set. Due to the definition of some class attributes it is possible that some attributes will not be filled.
Example 1: In the class ”Eenheid” within the domain ”Vastgoed”, there are three attributes that have this problem: ”bovenliggendeEenheid”, ”deelEenheden” and ”clusters”. To give some perspective on this problem, the name, type and description of the attributes can be found in Table 3.5

Table 3.5: Definition of three attributes that have a parent-child relation with one another.

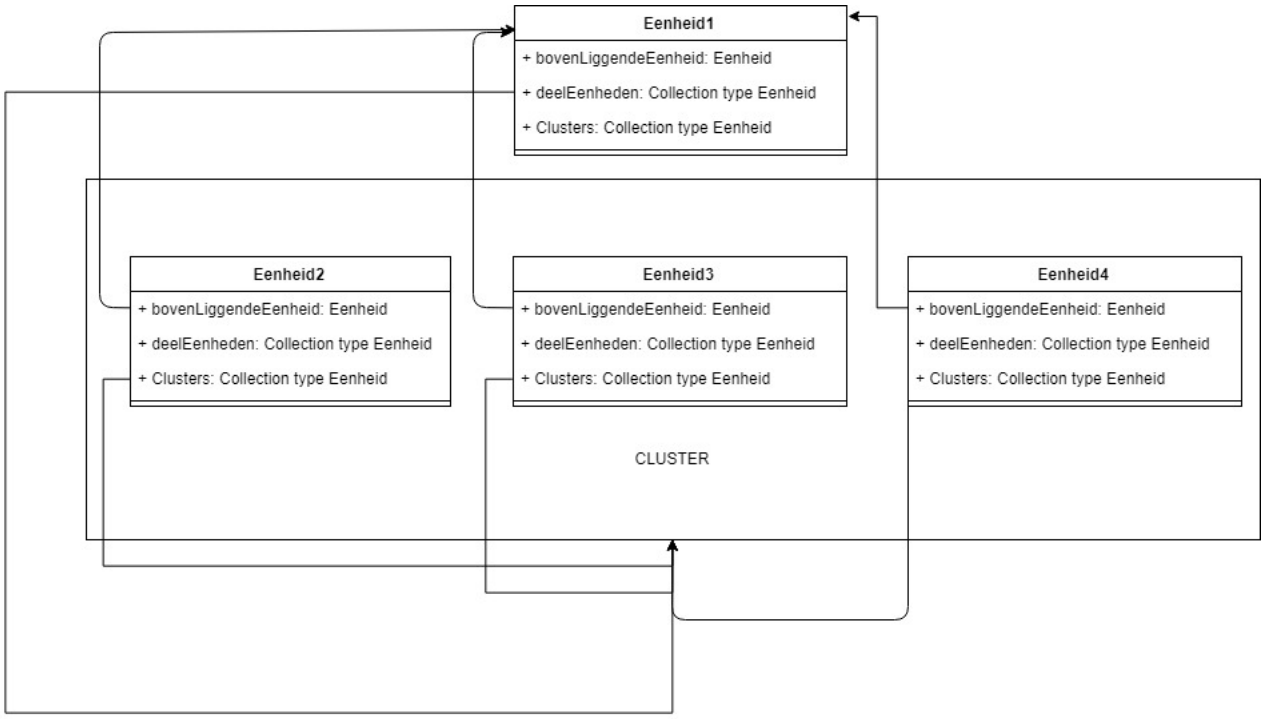
Name	Type	Description
bovenliggendeEenheid	Eenheid	The parent Eenheid, where the current Eenheid is part of.
deelEenheden	Collection type Eenheid	The instances of Eenheid, where this Eenheid is divided into.
Clusters	Collection type Eenheid	The clusters where this Eenheid is part of.

The definition in Table 3.5 makes it clear how the different attributes are specified. Attribute ”bovenliggendeEenheid” is the parent of this structure, the attribute ”deelEenheden” contains the children of a certain ”Eenheid” and ”Clusters” is a collection of the siblings of an ”Eenheid”. Figure 3.1 shows how an implementation of such a tree structure could look like. The exact tree depends on the implementation of the ”Eenheid”, but will have such a form. In this example the lowest level of ”Eenheid” instances has the attribute ”deelEenheden” empty, not referring to a collection of ”deelEenheden” because the collection does not exist. Not every implementation will be the same, but the example shows that there are attributes that are not filled with any reference. So not all attributes in VERA are filled.

Problem 2: Mapping two similar classes with different attributes per class (check attributes).
In the process of mapping the vendor model to the standard model, there were classes that might be the same. The only problem is that they could not be simply mapped one on one, as they both may have attributes could not be mapped to the other model.
Example 2: In the VERA class ”NatuurlijkPersoon” there is an attribute ”reacties” defining the requests to apply for a certain real estate object. In the vendor model a similar class exists, defining the responses on possible housing options. The classes represent somewhat the same, but there are differences in attributes. This introduces a problem of not being able to map two classes one on one.

Problem 3: Multiple mappings from a vendor to the standard model and/or the other way around are

Figure 3.1: Example of the structure of multiple instances of the class ”Eenheid”.



possible.

Between the classes of the vendor and the standard model are differences. These differences can be solved through mapping. The problem in this mapping is that a translation of a vendor class/attribute to a standard class/attribute can be done in multiple ways for some classes/attributes. There are multiple mappings possible from the vendor model to the standard model.

The multi-mapping can occur in two ways: one way where there is one attribute/class from the vendor and multiple attributes/classes in the standard model that can be filled with the vendor information, and the other way is that there is one attribute/class in the standard model and there are multiple attributes/classes that can fit in the vendor model.

Example 3.1: One vendor class to multiple standard classes, the vendor model has one class ”Complaint”. This class can be fitted in the VERA class ”Zaak” and the VERA class ”Signalering”. This means that there are multiple ways of mapping the vendor model to the VERA model.

Example 3.2: One standard class to many vendor classes, within the VERA class ”NatuurlijkPersoon” there is an attribute ”voorrangsregelingen” defining a list of the type ”voorrangsregeling”. It can fit multiple vendor classes. The vendor model has the class ”TenantUrgency” and the class ”TenantPriority” that fit the description of ”voorrangsregelingen”, it is important to make sure the right classes are mapped to one another.

Problem 4: Different identifier attributes in different versions of the standard.

The standard is continuously in development, to keep up with the changes in the market. In the past a standard model had different ways to identify an instance. Such identifiers are called key attributes, and these key attributes might change over time. It is important to keep track which key attribute is used in the current version of the standard model, so that the vendor model does not differ from the standard.

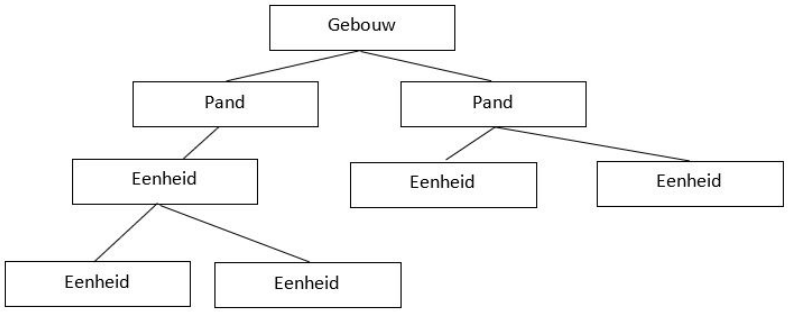
Example 4: As defined in the document ”Bijlage A - Standaarden en richtlijnen” in Section 2.1 ”GM11 Gebruik van sleutelattributen”, different key attributes have been used in the past, such as ”id”, ”bronsysteem” and ”businessKey”. Currently the attribute ”code” is defined as the class identifier [VER16a].

Problem 5: Class and attribute names differ between models.
The names of classes and attributes differ per model, meaning that a mapping cannot be made with only knowledge of the names of the classes and attributes.
Example 5: The VERA class ”Relatie” and vendor class ”Acquaintance” are very similar, as they describe the same entity. The difference is that both models have their own way of naming their classes and attributes, which makes it not possible to create a mapping with only the names of the classes and attributes.

Problem 6: Some attributes are generated for the standard model (VERA-generated).
Within the standard model there are different attributes that are generated specifically for the standard model. The problem is that there might be difficulties in mapping these with the vendor model. As the standard model attributes are not in the vendor model.
Example 6: VERA attributes like ”id”, ”bronsysteem”, ”businessKey” and ”code” are attributes that are defined in the VERA model. Mapping these attributes such that they are conform the VERA model is important, but also difficult. Such attributes do not exist in the vendor model or in another way.

Problem 7: Complex composition structures.
In the standard model certain attributes are defined so that they refer to other instances of classes, creating some sort of class structure. For instance a tree structure where a attribute can refer to a set of child-instances, these child instances can then also have children or parent instances. The structures are specifically defined within the standard model, therefore they might not be in the vendor model, or the structure exists in another way with other classes and references. Therefore it is difficult to map the vendor model to the VERA model.
Example 7: A tree structure can be found in the structure of the class ”Gebouw”, ”Pand” and ”Eenheid” within the domain ”Vastgoed”, where the class ”Pand” can consist of an ”Eenheid”, and an ”Eenheid” can consist of multiple instances of ”Eenheid”. A ”Gebouw” can be a synonym of the class ”Pand”. But it can also stand for a cluster of instances of ”Pand”. The ”Pand” instance can have on its own instances of the class ”Eenheid”. This structure of entities is visualized in Figure 3.2.

Figure 3.2: Example of a complex tree structure in the VERA model.



What becomes clear from this example is that the structure of these classes is explicitly defined within VERA and needs to be taken into account when a translation from a vendor to the VERA model is made, as it can become a complex structure of classes.

Problem 8: Some standard model attributes are not found in the vendor model.
Some standard model attributes could not be found in the vendor model. A reason could be that the available documentation on the vendor model was not detailed enough to find the vendor attribute that can be mapped to the standard model, or the attribute is not in the vendor model.
Example 8: In the process of mapping the vendor model to the VERA model, the VERA attribute ”Opleiding” was not found in the vendor model. It could be that due to too little information on the vendor model it was not found or it is not in the vendor model.

Problem 9: Identifier attributes in different class structures.
The standard model is not the same as a vendor model. So there are differences, such a difference can also be in the identifier attributes of the models. It can occur that the standard model has a specific class for something, but the vendor has it stored in an attribute of a class that is not the same as the standard model class. Then the standard class has this specific class with an identifier and a vendor has the needed information in an attribute of another class with not an identifier. How can the identifier attribute then be mapped or created, as it does not exist in the vendor model.
Example 9: In some vendor models address is an attribute of the class ”Tenant”, but in the VERA model address has its own class which is named ”Adres”. The VERA class has identifier attributes that specify each address, such an attribute is not for the vendor address as it only exists as an attribute and does not have any identifier attributes. How does a vendor implement the identifier attribute when mapping to the VERA model?
The defined problems in this section are the problems that were found during the mapping process of a vendor to the standard model. The problems are as abstract as possible to make sure they can also be identified in other vendor models, but the defined problems are not all the problems. Different vendor can face different problems and there are possibly more problems this research did not come across, that will be for future work. For the problems defined in this section different solutions will be proposed in Section 4.1. With these solutions the guidelines to implementing a standard model on top of the vendor model will be given.

3.3 Data

The VERA documentation consists of several parts that describe the VERA concept. This documentation goes into the concept, principles, background, implementation and maintenance of the standard. The documentation also consists of several logical data models and explanation. These models enable users to understand VERA and implement it. The models are divided per domain. The following domains are specified in VERA:

- General (Algemeen)

- Relations (Relaties)
- Real Estate (Vastgoed)
- Agreements (Overeenkomsten)
- Maintenance (Onderhoud)
- Living space distribution (Woonruimteverdeling)
- Financial administration (Financiën)
- Dossier (Dossier)

For each of these domains corresponding classes and attributes have been specified. One can build an application upon these classes.

When building an API for the VERA model it is important to take all these classes into account, to make sure that all the different data attributes can be extracted. For the research an Excel file with all the classes and attributes had been made available. The Excel file consists of the following columns: File name, Class Domain, Class, number in class, Name, a column with "\N" character, Type and Description. For some classes within the Excel worksheet there was also an extra column with values that varied per class. For the purpose of creating the API the following columns were used: Class Domain, Class, number in class, Attribute Name, Type and Description. After the removal of the other columns the Type column was modified. The Type column consists of a primitive type or a VERA related type, these types can also be in a collection (collectie) of types. If there is a VERA type, the column has a HTML reference to the file where the type would be in with a `<a href="URL">` tag.

Table 3.6: Example of a row in the CSV file.

Class Domain	Class	#	Attribute Name	Type	Description
org.vera.relaties	Relatie	6	adressen	collectie RelatieAdres 	De adressen behorend bij de relatie.

An example of such a row can be found in Table 3.6. For the purpose of creating the API this tag is not needed, so the Type column was modified for all the VERA classes as follows:

1. Removed `<a href="`
2. Replaced `.html">`, with ":"
3. Removed `</a>`

After this was done the different domains that were bundled together in the Worksheet where divided into different CSV files. So that the domains could be used in the API creation per domain. For instance to check

the different data types within a certain domain, with a Python script. Using a Python script the unique data types were extracted. With the different data types a few errors were visual in the data. The errors were typos in the Excel sheet or missing values. This was solved by finding out what the correct data types were by reading the VERA documentation.

After the data modification process in the Excel sheet was done the existing types are analyzed.

3.3.1 Types

In the data extracted from the Excel worksheet there are different types for all the attributes within the classes. To make sure a supplier can easily adapt the API to their preferred programming language, it is important to change the primitive types in the VERA worksheet to the primitive types that are defined within the desired programming language. The Python script that can be found in Appendix A was built to extract all the different types from the excel sheet. The types can be divided into primitive data types and VERA related types. The VERA related types are defined by a prefix in their class definition, which is ”org.vera.”. The distinction between the primitive data types and the class related data types was made by filtering out the types that contain ”org.vera.”. This results in a list of primitive data types:

- base64
- boolean
- date
- datetime
- decimal
- DEPRECATED
- duration
- integer
- string
- uri

In this list also stands the data type DEPRECATED, which is a data type that indicates that the attribute of this type is not continued in the use of the standard class model. In the implementation of the API the type was commented out, to make clear that it did exist but is not used anymore.

3.3.2 API design

After the identification of the different primitive attributes, the API framework was chosen. The Microsoft .Net Web API framework [Corb] was chosen, as the choice of framework did not matter in a technical perspective and the person designing the API had experience with this framework. The .Net Web API tutorial [Cora] was used as a guideline. The tutorial describes a simple ToDo API with CRUD operations, which creates a service that can create, read, update and delete (CRUD) ToDo's in a web service. The different files in the API were identified, which consisted of the following: Context, Classes and Controllers. These files were analyzed and checked how an implementation with the VERA classes would come together. The choice was made to use the Excel file to generate all these files. The class files were generated per domain, with a corresponding name space. Also a Context file was generated that defined the different classes in a Database Context.

For the controller files one approach is to create an API endpoint for every class within the VERA model. Every API endpoint will consist of the CRUD operation, enabling a vendor or other system to Create, Read, Update and Delete an instance of every class. The Python script was further developed and in the end it was able to create the database context file, class files and a controller file for every class within the VERA model.

The second approach for the controller files is to create API endpoints for important classes, so called hub classes. These hub classes are specified through the number of references to them from other classes. A higher number of references indicates a greater importance of a class in the VERA model. In this approach for the ten classes with the highest number of references we have created API endpoints. These API endpoints also have the CRUD operations (create, read, update and delete), and are somewhat denormalized as the hub class attributes can be used in the hub class CRUD operations and do not need to be accessed through their own API endpoint. The Python script that makes this process possible is not strictly bound to only ten hub classes, this number is taken as an example.

With the described method two approaches towards a VERA API that can be implemented by the different vendors were defined. Next in the evaluation some solutions for the mapping problems will be given and the two APIs will be examined to make sure they can be implemented by the vendors.

The Python script that is developed for this research can be retrieved on request at the author.

Chapter 4

Evaluation and Solutions

In this chapter we will evaluate the different mapping problems and try to find solutions for them. The different solutions for the mapping problems from Section 3.2 can be found in Section 4.1. After the evaluation of the mapping problems and solutions the chapter will go into the generated APIs and the mapping of the vendor model will be tested to check if the APIs are able to be mapped to the vendor model.

4.1 Mapping solutions

In Section 3.2 the different problems that can occur when defining a mapping from a vendor model to the standard model are explained. In this section possible solutions for these problems will be given. The solutions of these problems are defined using existing literature and logical analysis. The problems are categorized into three categories:

1. Information only available in the standard model.
2. Information only available in the vendor model.
3. Information available in both models.

With the solutions given per category a clear overview can be maintained.

4.1.1 Information only available in the standard model

The solutions defined in this section are for problems that are caused by some form of information available only in the standard model. First the name of the problem is given followed by a possible solution.

For the two problems within this category the solution is somewhat the same. A vendor does not have a class or attribute. Therefore the problems can be solved with the same solution.

Problem 1: The standard model has an attribute that the vendor model does not have.
Problem 2: The standard model has a class that the vendor model does not have.
Solution 1.1: Set the attribute or class to not existing.
If the vendor does not have the opportunity to retrieve the attribute or class from the vendor model or in some other way, the vendor can only choose to set the attribute or class to a value that indicates that the attribute or class is not existing. In most implementations this will result in setting the value of the class or attribute to NULL, as this is a common practice to indicate that the class or attribute is non existing, it might differ in some implementations. So it is important to set the attribute or class according to the implementation.
Solution 1.2: Create a new attribute or class in the vendor model.
The vendor can choose to create a new attribute or class in its model to assure its compatibility with the standard model. Then the attribute or class needs to be filled with the corresponding data. The supply of data can depend on the end-user who has to fill the required data, or in some cases it can be extracted using existing data within the model.

4.1.2 Information only available in the vendor model

The solutions defined in this section are for problems that are caused by some form of information available only in the vendor model. First the name of the problem is given followed by a possible solution.

For the two problems within this category the solution is somewhat the same. The standard model does not have a class or attribute. Therefore the problems can be solved with the same solution.
Problem 1: The vendor model has more attributes than the standard model.
Problem 2: The vendor model has a class that the standard model does not have.
Solution 1.1: Use a second layer for data exchange.
A more vendor specific solution is to create a second layer to exchange data. It is a best practice method already used in the industry. This extra layer will consist of attributes that are missing in the standard model, but are needed for some other party. The attribute is not officially part of the standard model anymore, but a new vertical or horizontal sector model is defined. The extra layer is a way to create vendor specific attributes and classes that can be exchanged.
Solution 1.2: Add attributes (VERA specific).
This solution is very specific for this standard model. In VERA data model there is an attribute ”extraElementen”. It’s origin comes from the StUF model [EGE18] as a way to add data that is not already in the standard model, such as VERA. StUF is a standardization initiative for data exchange by the Dutch government to embrace data standardization in IT systems that collaborate with the Dutch government systems. VERA is also using parts of the StUF initiative, one being the ”extraElementen” attribute. An attribute that does not exist in the VERA model can be added through the ”extraElementen” attribute, but this attribute is only meant for new attributes that will be sector-wide. To define a sector wide attribute, different agreements about the

definition have to be made. If the attribute is only missing for one vendor the ”extraElementen” industry-wide solution is not suited for this problem and then Solution 1.1 is a better option for the missing attribute or class.

4.1.3 Information available in both models

The solutions defined in this section are for problems that are caused even if the information is available in both data models. First the name of the problem is given followed by a possible solution.

Problem 1: Not all attributes will be filled.
Solution 1: Clear not filled definition.
To prevent problems with unclear unset variables a solution is to make clear agreements on how a non filled variable will be defined. An example of such an agreement is that if a variable is not filled it will be set to a certain value. In most programming languages this value will be NULL, indicating it is not referenced to anything. Some development teams choose to do it another way, with not setting a variable. Due to the different practices in this spectrum, clear agreements on non filled variables are a good solution.
Another solution is that the organization governing the standard defines a default value for the standard model, in this way it will be clear for everyone using the standard when an attribute is not filled. In that way the development team of a vendor is not responsible, but it will be a responsibility of the standard.

Problem 2: Mapping two similar classes with different attributes per class.
Solution 2: Check for similarities between classes.
A solution for this problem is to put the two models next to one another and check which attributes are similar and which are not. To execute such a solution, clear documentation on both the models is a requirement. Otherwise, definition problems in the mapping can occur.

Problem 3: Multiple mappings from a vendor to the standard model and the other way around are possible.
The problem has two perspectives, but a solution for both perspective is sufficient.
Solution 3: Better understand the models.
A very straightforward solution is to have more knowledge of both the models. With enough knowledge of both models it is possible to create a mapping that is correct. Both models will be compared and the right classes and attributes can be mapped from one model to another, thus solving the problem of multiple mappings.

Problem 4: Different identifier attributes in different versions of the standard.
Solution 4: The standard is continuously in development, resulting in different versions of the standard. To make sure the latest identifier attributes are used, the latest version of the standard must be used. Another possibility is to use API versions, that way certain API versions support certain identifier attributes.

Problem 5: Class and attribute names differ between models.

Solution 5: Some class and attribute names differ per model. To map the models to each other information is needed. In the case that names differ, the information on the definition of the classes and attributes is needed. If the definition is clear, a mapping can be made.

Problem 6: Some attributes are generated for the standard model.
Solution 6: Attributes generated for the standard model will be defined in some sort of documentation, so understanding and using this documentation is a good way to solve the problems.
In the VERA case, the document "Standaarden en richtlijnen" [VER16a] describes how to implement VERA generated attributes. For mapping purposes the required information can be found in this document, enabling a way to generate the attributes conform the VERA guidelines.

Problem 7: Complex composition structures.
Solution 7: This is an implementation problem, and therefore the solution lies in understanding the vendor model enough to be able to make a correct mapping from the vendor model to the standard model. Then it is possible to make a mapping from a complex structure to another complex structure. The complex structure problem is not only for complex tree structures as the title suggests, there are also other complex structures that could be found in a standard model, such as a associative array.

Problem 8: Some standard model attributes are not found in the vendor model.
Solution 8: When attributes are not found in the vendor model, it could be that the attributes is not in the vendor model. If the attribute does not exist the solution proposed in Section 4.1.1 can be used. In some vendor models it might be possible to combine multiple attributes to create the required standard model attribute, for some vendor models this could be a solution.

Problem 9: Identifier attributes in different class structures.
Solution 9: In the given case the attributes is not in the vendor model, but are needed to deliver standard formatted data. There are two possible solutions both of them involve making agreements on generating the identifier attributes. The first solution will be on the sector level, where the standard model will define an agreement of how a key attribute needs to be generated. For instance that all the key attributes will be defined using class name and part of the attribute name. The second solution will be where the vendor will decide how their key attributes will be defined, parties using their translation to the standard can then inform how their key attributes are defined to use the attributes in their systems.
In case of VERA the document on the guidelines of VERA [VER16a] describes in its Section 2.1 GM09 that all the attributes within the VERA model are optional, to assure that every system that wants to use the VERA standard can correctly form their messages with VERA. With this choice it is the responsibility of the receiving party, which is the client of the vendor, to define which attributes need to be filled. Therefore the solution is to make agreements with the vendor and the client to define how the key attributes will be generated.

What becomes clear from these different solutions is that there are many problems that can occur when

mapping a vendor model to a standard model, but all the different problems can be solved. None of the defined problems lack a solution, so mapping the vendor model to a standard model is possible. However, something being possible does not means it is an easy task. As described with the problems there are a lot of obstacles that need to be overcome. The solutions proposed here do not solve all the problems, as there are probably more problems in the mapping process that the research did not come across.
Automated solutions might be possible, but they are problematic as the different problems and solutions are vendor specific. With sufficient domain knowledge of the vendor and VERA model one might define an automated solution, but it will be a vendor specific solution, because of the differences between vendor models.
During the process of mapping and finding solutions, it was very clear that having sufficient knowledge of both the models is key to create a mapping. All the defined problems have in some way a solution where knowledge of the models plays a role. From stating if an attribute exists in a vendor model to modeling complex structures, in all cases knowledge on both models is important.
The level of detail of the classes in VERA is very high. Almost any vendor can map to the VERA model, but many VERA attributes do not exist in the vendor model. Such a level of detail (normalization) makes it difficult to create a mapping to the VERA model, as a vendor can not fill the whole VERA model. Such a problem does not have a specific solution as described earlier, but another possibility could be to define certain classes that are more important than other classes. Then the focus will be on the important classes that vendors should implement instead of implementing a whole model, that has many attributes and classes that cannot be implemented. These important classes could be named *hub classes*, that work as some sort of hub. Their importance could be based on the number of references to them or, in a working example, how often they are being used. Of course what constitutes an important VERA class could also be based on (VERA) domain knowledge and provided as part of the VERA documentation, but we wanted to propose an automated criterion here. Using the hub classes the vendors can become compliant with the standard model by first implementing the hub classes. After the implementation of the hub classes vendors can become more compliant by implementing other classes that are also important for the vendors' business. This would result in a vendor adaptive VERA API.

4.2 API and Test mapping

As described in Subsection 3.3.2 two APIs have been developed using the existing VERA information. The first one is an API that has API endpoints for all the VERA classes and the second one is an API that implemented the ten most important VERA classes, also called hub classes. In this section we will combine the solutions for the mapping problems and the two API approaches to define how a vendor can implement such an API.

4.2.1 All classes API

This API has been developed to create API endpoints for all the different classes within the VERA model to enable every vendor to implement the API, the problem with this API is that most of the vendors do not have all the classes that are in the API, so they cannot map the whole API.

The vendor can solve this by mapping its data model to this API using the given solutions for the problems that have been found in the mapping process. These solutions can be found in Section 4.1 With this API the vendor can deliver its data in the format of the different VERA classes. A test mapping to this API looks the same as the tables that have been defined in the process of mapping, but due to confidentiality reasons for the vendor these tables are not in this thesis.

4.2.2 Hub class API

This API has endpoints for the ten most important classes within the VERA model. This choice was made to ensure that all the endpoints can be implemented by the different vendors. From this point the vendor can choose to expand the API endpoints with classes that are important and can be mapped from their system.

The vendor can map its data model to this API using the given solutions for the problems that have been found in the mapping process. With this API the vendor can deliver the ten most important VERA classes, and in the future extent the API to implement more classes besides the ten classes. This can be done through adding controllers that serve as API endpoints for the classes they want to implement in the Python script that generates the files, thus becoming more VERA compliant. A test mapping to this API looks the same as the tables that have been defined in the process of mapping, but due to confidentiality reasons for the vendor these tables are not in this thesis.

The API only has endpoints for the hub classes, but is defined using other classes that can not be accessed through class specific API endpoints. On these classes the CRUD (Create, Read, Update and Delete) operations can be used through the hub classes, because the not hub classes are defined in the hub class API endpoints.

The generated APIs are gateways to becoming VERA compliant, but the work lies in the hands of the vendors. The vendors can implement these API options for their data models, such that they can deliver their real estate information in the VERA format.

4.2.3 API requirements

In the Chapter 1 some requirements were given to design the API. These requirements were needed to make sure the API could be adopted by the different vendors. So to conclude the API design these requirements will be discussed how they were defined in the API design. The following requirements were defined:

1. The API is implementable for vendors, the mapping between data model and the API methods can reasonably be made. (The API must be sufficiently denormalized.)

2. The API is generic: the whole object graph can be traversed and manipulated with the API.
3. The API is simple: it is consistent over classes, which makes it easier to use for developers.
4. The API can be extended for data not part of the VERA model.

Now we will go into each requirement and describe how they are fulfilled in the API design.

The API is implementable for vendors, the mapping between data model and the API methods can reasonably be made. (The API must be sufficiently denormalized.)
The mapping process can be done, because the API defines all the different classes and attributes from the VERA model. Also the vendor can modify the API that it has the API endpoints the vendor needs. The mapping between the models can be made as the different problems and solutions in the mapping process have been defined. On the documentation level the API is defined based on the VERA classes and attributes, so the VERA documentation is sufficient.

The API is generic: the whole object graph can be traversed and manipulated with the API.
In both the API design approaches the API is designed using the VERA model using a CSV file that contains all the VERA classes and attributes. In the entire VERA API approach the whole object graph can be traversed and manipulated. For the Hub class approach all the implemented classes are generated from the CSV file, so that part of the object graph can be traversed and manipulated with the API.

The API is simple: it is consistent over classes, which makes it easier to use for developers.
For the API endpoints the possible operations that can be used on the API are consistent, because the API has only CRUD operations on each endpoint. CRUD operations are broadly used in the world of API design, so developers are used to it. This makes it easy to use for developers.

The API can be extended for data not part of the VERA model.
The APIs can be extended as they are generated using a CSV file. If a new class or attribute is added to the CSV file, the Python script can generate the new API files, which adds the new data to the API design. Therefore the API can be extended.

With the fulfilling of the requirements an API was designed that can be implemented by the different vendors and hopefully enable them to shift to a VERA compliant system.

Chapter 5

Discussion

The initial goal of this research was to create an tool that will automatically generate a VERA API that enables a vendor to almost automatically map its model to the VERA standard. This API will enable the vendor to translate its proprietary data model into the VERA model, than the vendor is able to exchange data seamlessly with other vendors. This removes the need for custom data modification, which can lead to cost reduction. The goal was to create a tool such that this would be done almost automatically. In the end it became clear that such a tool was very difficult to built as the problems are very vendor specific and domain knowledge of the vendor model was needed, therefore the research focused on other points from where further development can take off.

So the research focused on identifying the different problems within the mapping process, finding solutions for these problems and generating an API based on the VERA model, that a vendor can map its model to. The proposed automated API was not possible to be made, because the problems turned out to be vendor specific and there was no automated generic solution possible for all vendors. To enable future development a method is developed to generate an API automatically based on the VERA meta data. The API does not solve the mapping problem, because there is still a mapping mismatch. But it is a first step that will:

1. save vendors time as the VERA side of the problem is already coded.
2. give the VERA model a platform for automated operations that simplifies a mapping, and can generate automatically a new API.

The VERA model is a very specified model with many specific classes and attributes, many vendors do not have such specific attributes and classes. This makes it hard for a vendor to deliver all the different VERA classes and attributes. To solve this a approach to create a denormalized API was proposed, but this was hard as there was no clear distinction between classes that could be used for denormalization. In this research the example was to take the 10 classes with the most references to them, making them more important than other classes. This approach can also be based on class diagram metrics which is interesting for future research, Osman et al. use in their paper Machine Learning algorithms to predict key classes in class diagrams [OCvdP13].

There are also other ways to define key classes in a data model, that can be looked into in future research. The denormalized API is an interesting option to look into as it enables vendors to more easily adapt the VERA standard to their model. Another approach on creating a denormalization is that the VERA organization defines key classes or required classes, which gives vendors a starting point to start from.

Overall the research was able to provide first steps to defining an API that can enable vendors to deliver their data in the VERA model. Next up for the VERA standard is to look into future research.

Chapter 6

Conclusions and Future Work

The title of this research is ”Aligning an industry data model with applications in the real estate”, where the industry data model is the VERA data model and the applications are from different vendors in the real estate industry that want to adapt the VERA data model. The aim was to align the two models, to enable the different vendors to adapt the data model.

For the adoption of the VERA data model the proposed solution was to built an API that the different vendors can map to. With this API the vendors would be able to deliver their data in the VERA format. To support the vendors, the different problems that can occur when mapping a vendor model to the standard model were analyzed and for each problem a solution is proposed.

The mapping of the vendor model to the VERA was done by hand, with the available information on both the models. Where the VERA model was taken as output and the vendor model, who delivers a ERP application for the real estate industry, was taken as input. In this process similar VERA attributes and classes were searched in the vendor model, if some attributes or classes were found they could be mapped to the VERA model. With this process several problems in the definition of the class structure or naming of attributes were analyzed. For the API design process two approaches were used: the first approach was to create an API with endpoints for all the VERA classes; the other approach was to define endpoints for VERA classes that are more important than other classes, their importance was defined on the number of references from other classes to their class within the VERA model. This is one simple example of a more automated solution. For the API design an API best practices example from Microsoft was used.

Each problem that was found in the mapping process can be solved, this thesis defines for each of them a possible solutions that can be used when implementing a mapping between the vendor and the standard model. The API that was designed fulfills the proposed requirements which are that the API: is implementable, is generic, is simple and can be extended. For the API design approaches it was possible to map the vendor model to the API. The solutions to the mapping problems enable the vendors to be able to implement the API correctly.

With these solutions a first step into solving the mismatch of vendor model and standard model is taken, but there is more to this problem. From the analysis and the solutions it became clear that domain knowledge of the vendor and the standard model is required, which makes it hard to define automated solutions. These automated solutions are important to define a adaptive solution for all the vendors, and needs to be further explored in future research.

To conclude, from this research an API specification can be generated that fulfills all the different requirements. With the mapping problems and solutions it will enable the different vendors to implement an API that will deliver their data in the VERA format.

6.1 Future work

The problems that we found in the mapping process are not all the problems. It will be interesting to look what problems the vendors come across when implementing such an API. It will be interesting what problems are vendor specific and what problems are related to the standard model or API.

In the related work some other Industry Standard Data Models (ISDMs) were described, some best lessons can be learned from how these models function in their industry. In what fields do these ISDMs excel and what are common pitfalls for them? Data model transformation will be particularly an interesting subject when looking into these ISDMs.

The VERA model classes and attributes are all optional to enable vendors and other parties to implement the model with as less restrictions on the model as possible, such that almost anyone can implement it. In the second API design approach we chose to define hub classes based on references, it might be interesting for the VERA standard to define certain hub classes in their documentation. That way the VERA model could define certain VERA compliance levels, that vendors can use to identify their products with.

For the denormalization of the API it will be helpful to look into the different possibilities of class metrics, that define the use of classes and their importance. Osman et al. does this on a class diagram level, but this might also be reused for the VERA model [OCvdP13].

With these possibilities future research can be set up to support a better alignment of an industry with an industry data model.

Bibliography

[Ass] PODS Association. Pipeline open data standard (pods). https://www.pods.org/. [accessed: July 2018].

[Ass10] Schools Interoperability Framework (SIF) Association. Schools interoperability framework SIF implementation specification 2.4. June 2010. [accessed: July 2018].

[Cora] Microsoft Corporation. Create a web API with asp.net core and visual studio for windows. https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-2.1. [accessed: June 2018] [Dutch].

[Corb] Microsoft Corporation. .net framework web. https://www.microsoft.com/net. [accessed: June 2018] [Dutch].

[EGE18] EGEM. Standaard uitwisselings formaat StUF 03.01. https://www.gemmaonline.nl/images/gemmaonline/f/fa/Stuf0301.pdf, February 2018. versie 27, (EGEM).

[fIEC17] Directorate-General for Informatics (European Commission). *New European interoperability framework promoting seamless services and data flows for European public administrations*. Publications Office, Luxembourg, 2017.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1994.

[KL18] H. Köhler and S. Link. SQL schema design: Foundations, normal forms, and normalization. *Information Systems*, 76:88–113, 2018.

[LKR11a] K. Lano and S. Kolahdouz-Rahimi. Design patterns for model transformations. *ICSEA*, 2011.

[LKR11b] K. Lano and S Kolahdouz-Rahimi. Solving the TTC 2011 model migration case with UML-RSDS. *arXiv preprint arXiv:1111.4741*, 2011.

[OCvdP13] M. Osman, M. Chaudron, and P. van der Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *2013 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 140–149. IEEE, 2013.

[Ram03] R. Ramakrishnan. *Database management systems*. McGraw-Hill, Boston, 2003.

[VER] Stichting VERA. Vera publicaties - VERA 3.1. https://www.stichting-vera.nl/publicaties/vera-3-1. [Dutch].

[VER16a] Stichting VERA. Bijlage a - standaarden en richtlijnen. Technical report, 2016. [Dutch].

[VER16b] Stichting VERA. Bijlage b.2 - gegevensdomein relaties. Technical report, 2016. [Dutch].

[VER16c] Stichting VERA. Bijlage b.3 - gegevensdomein vastgoed. Technical report, 2016. [Dutch].

[VER16d] Stichting VERA. Vera 3.1 - hoofddocument - definitief - 20160616. Technical report, 2016. [Dutch].

Appendix A

VERA types filter

```
import csv
import pandas as pd
import numpy as np

#reading the CSV files into the dataframes
df_algemeen = pd.read_csv('veraClassModel-algemeen.csv', sep=';')
df_dossier = pd.read_csv('veraClassModel-dossier.csv', sep=';')
df_financien = pd.read_csv('veraClassModel-financien.csv', sep=';')
df_onderhoud = pd.read_csv('veraClassModel-onderhoud.csv', sep=';')
df_overeenkomsten = pd.read_csv('veraClassModel-overeenkomsten.csv', sep=';')
df_relaties = pd.read_csv('veraClassModel-relaties.csv', sep=';')
df_vastgoed = pd.read_csv('veraClassModel-vastgoed.csv', sep=';')
df_woonruimteverdeling = pd.read_csv('veraClassModel-woonruimteverdeling.csv', sep=';')

#Get list of types
arr = []
arr.append(df_algemeen["type"].unique())
arr.append(df_dossier["type"].unique())
arr.append(df_financien["type"].unique())
arr.append(df_onderhoud["type"].unique())
arr.append(df_overeenkomsten["type"].unique())
arr.append(df_relaties["type"].unique())
arr.append(df_vastgoed["type"].unique())
arr.append(df_woonruimteverdeling["type"].unique())

veraTypes = []
```

```
for array in arr:
    for vType in array:
        vType = vType.rstrip()
        if "org.vera" not in vType:
            veraTypes.append(vType)

veraTypeFrame = pd.DataFrame()
for i in veraTypes:
    veraTypeFrame = veraTypeFrame.append({'types' : i}, ignore_index=True)

#Get rid of the duplicates in the dataframe
veraTypeFrame.drop_duplicates(inplace=True)

#write to csv file
veraTypeFrame.to_csv(path_or_buf="test.csv", index=False)
```