A model driven approach to modernizing legacy information systems

Author:
Sander Goos
S0113409

Supervisors:
Dr. Ir. M. van Keulen
Dr. I. Kurtev
Ir. F. Wijnhout
Ing. J. Flokstra

Master Thesis

University of Twente

in collaboration with

Thinkwise Software Factory B.V.

November 2011

A model driven approach to modernizing legacy information systems

A thesis submitted to the faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, the Netherlands in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

with specialization

Information Systems Engineering

Department of Computer Science,
University of Twente
the Netherlands

November 2011

Abstract

We propose a general method for the modernization of legacy information systems, by transforming these systems into model driven systems. To accomplish a transformation into a model driven system, first a model is extracted from the legacy system. This model is then transformed into a model driven system, using Model Driven Engineering. This means that for the transformation, a model is constructed and an MDE tool is used to generate the executable transformation code for it. The method is not limited to the data-model of the legacy system, but is instead applicable to the entire system. Furthermore, the method has a best-effort character, and allows for automatic traceability. By means of a pilot modernization project, the method is validated with the Thinkwise Software Factory as the MDE tool.

Acknowledgements

I would like to thank a few people who helped me during the course of this project, and have made it an interesting time for me. First of all, I want to thank my supervisors: Maurice, Frank, Ivan and Jan. The meetings we had were always really motivating and it was a pleasure to work with them. Maurice, in particular, has taught me a lot during the project, and was enthusiastic about the project from the start. I also want to thank my employer Thinkwise, and in particular Victor, for the great amount of confidence in me and for the freedom I had during the project. I thank my colleagues at Thinkwise, and my fellow students on the third floor, who contributed to a fun and productive working environment. I thank my girlfriend, Angelique, who has been very patient and supportive during busy times. Finally, I thank my family and friends who have always supported me during the course of my study and have motivated me to pursue.

Contents

Chapter 1

Introduction

Imagine a classroom full of physics students, waiting for their first lecture to commence. The teacher closes the door and walks to the front of the classroom. "Good morning class, today's lecture is about mechanics.", he announces. "A very important scientist in this field was Sir Isaac Newton. According to my colleagues, he developed some exciting and revolutionary laws – and it is all written down in this book." The teacher holds up the Principia Mathematica from 1726. "Unfortunately however," he continues, "it is all written in Latin!". Since Latin is not part of the curriculum, the students all look confused at each other, and one asks: "Do we now first have to learn Latin?". "No, don't worry..." the teacher replied, "...I don't understand Latin myself either. Besides, when something is so old that it is written in Latin, how can it possibly apply to today's world?! No, I think it will be better if we create our own, more modern, theories. Will you all join me to the apple tree in the garden? I am sure it wouldn't be too hard to figure it out...".

What do you think of the approach of this teacher? Absurd? Naive? We think most people would agree that the teacher greatly underestimates the redevelopment of the theories, and that translating the old works would be a lot wiser. But when this seems so obvious, then it might be surprising that in software modernization projects, redevelopment is not uncommon. Legacy computer systems often are built with architectures and (programming-)languages no longer used or learned by programmers. While the knowledge in these systems might not be so unique as Newton's Principia Mathematica, a lot of systems have taken years of development – and therefore do represent great value. When these systems become too costly to maintain, or when new technologies need to be incorporated, they need to be replaced with modern variants. However, since the modern programmers do not fully understand the legacy system, not rarely, it seems easier to redevelop the system from scratch in a new language.

In [1], Ulrich states that the knowledge captured in the legacy system should serve as an important resource in modernization projects. However, extracting and using that "knowledge" from a legacy system also requires effort. Therefore, the author provides means for building a business case for the transformation of legacy systems. The business case should be used to determine whether

transformation is the right strategy in a modernization project. By transforming parts of the original system, the value and knowledge incorporated in them can be *recycled*. Since the target platform typically also has more technological advantages than the source platform, we refer to this process as: *Upcycling*. Upcycling in general is a form of recycling (waste materials / useless products) where the result is of a higher quality than the original product. In our case, the original product is a legacy system and the result is a new system on a new platform. The question of how to do this in a general way is interesting, and the question to what extent we can automate this, even more. In this thesis we answer these questions by developing such a general method and validate it with a prototype.

The following sections give a general introduction to the thesis. First, the context of the thesis is discussed, after which the problem is elaborated. When the problem is clarified, we advance in stating the focus by formulating the goals of the project. Then, the method of validation is treated, and we finish the chapter with the contribution of the thesis.

1.1 Context

The problem context in this thesis is that of legacy information systems and Model Driven Engineering, or MDE. The legacy system is our source system and serves as the initial starting point. The goal is to transform this system in such a way that it can be maintained and developed further with MDE. In the remainder of this thesis, such a system is referred to as a model driven system. The following subsections serve as a basic survey of these concepts and introduce the terminology that is used in the remainder of the thesis.

1.1.1 Legacy systems

Since a central subject in our research is a legacy information system [2], an explanation about what we mean by that is in place. The definition below for a legacy system is borrowed from Wikipedia [3]:

> *A legacy system is an old method, technology, computer system, or application program that continues to be used, typically because it still functions for the users' needs, even though newer technology or more efficient methods of performing a task are now available.*

In our work, we look at legacy systems that need to be replaced. There are various reasons why a company wants to replace legacy systems, for instance:

- Maintenance costs get too high.
- A new module is needed, but there are no programmers any more in the company that know enough about the system to be able to modify and extend it.
- The company wants to create a new interface (for instance for the web) to the system, but the legacy system is not designed to support that.

- The legacy system does not scale well enough with the growth of the company.
- It is not easy to connect or integrate the legacy system with other systems in or outside the company.

The legacy systems on which we focus, are legacy *information* systems. Systems with their primary purpose being the storage and retrieval of business data. This does not mean that these systems do not perform operations or computations. Instead, the majority of legacy information systems typically have a considerate amount of business logic contained in them. It is just that their *primary* goal is to store and retrieve information – which is why we call them legacy information systems. We transform these systems into model driven systems, using Model Driven Engineering.

1.1.2 Model Driven Engineering

Model Driven Engineering [4] is a discipline that followed from OMG's Model Driven Architecture, or MDA [5, 6]. MDA is a method where models are extensively used in the design of software systems. As can be seen from other branches, models can provide crucial insights in the system before it is implemented. They serve as a clear guideline for the implementers and also allow to detect possible faults early.

Before we continue, let us clarify what we mean with a model. In most design projects, a model is used to represent the system or building that is not yet created. In construction, for instance, a cardboard architectural model is created before the construction of the building begins. This model, which is a lot smaller than the actual building, provides great insight in what the actual building will look like. Next to a physical model, a mathematical model of the construction can be very useful too. Such a model can be used to, for example, predict stability issues. In software engineering, this is no different. A model represents the design of the system often in a more comprehensible abstract form than the system itself, and can also be used to make predictions about the system. Aside from the different aspects being modelled, a major difference in the physical architectural model, the mathematical model, and the software model, is the language used. The physical model has the "language" of cardboard and glue, the mathematical model: mathematics and the software model, for instance: UML (Unified Modeling Language).

The definition of a model from the MDA Guide [5] is as follows:

> *A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.*

In MDE, the language in which a model is created can, on its turn, also be defined with a model. This second model is then referred to as a meta-model. Hence, a meta-model defines the abstract syntax of a modelling language. A model that uses a certain meta-model is said to *conform to* that meta-model. This means that every construct used in the model, is defined in the meta-model.

A model only means something to a person when its meta-model is also known. For example, when someone never learned mathematics, a mathematical model makes no sense to him or her. Also, a construct in one meta-model can have a different meaning in another meta-model. This makes a model inextricably connected with its meta-model.

In earlier approaches to modelling in software development, the need keep the models up to date with the system, diminished once the system was operational. Therefore, these models are bound to get out-of-sync, and run the risk of becoming legacy themselves. This can make them practically unusable in modernization projects. In Model Driven Engineering, or MDE, the model is actively used in the creation and maintenance of the system. In fact, the model is the *primary* place where changes are made to the system, i.e. the models are treated as first class entities. The entire system is generated or interpreted directly from the model describing it, making the model a complete definition of the system. The benefit of MDE is that the model and the system can no longer get out-of-sync, since they are tightly connected. The usefulness of this principle gets larger when the system gets larger. Imagine you need to change a certain behaviour of a large system when years of maintenance has passed since the first release. When there is a model of the system available, of which it is *guaranteed* that it is in sync with the system, i.e. the system conforms to that model, this can be of much help in finding the appropriate modules, predict impact, and so on.

In the remainder of this thesis we use the term: MDE tool, to refer to the preferred tooling or development environment for Model Driven Engineering. We assume that the MDE tool uses a fixed meta-model, to which all the model need to conform to.

1.1.3 Model transformations

Since models are the primary elements of development in MDE, model transformations are very important. In the broadest sense, a model transformation is a process in which an existing model is used to create another model. Model transformations come in different forms.

We adopt the model transformation taxonomy of Mens et al. [7]. In the taxonomy, model transformations are categorized into two dimensions which are shown in table 1.1. The first dimension separates horizontal model transformations from vertical model transformations. With horizontal model transformations the level of abstraction does not change during the transformation, in vertical transformations it does change. The second dimension of the model transformation separates endogenous transformations from exogenous transformations. An endogenous transformation is a transformation where the meta-model of the source and target model are the same. An exogenous transformation is a transformation where the source model has a different meta-model than the target model. Both dimensions are orthogonal. Table 1.1 shows the four model transformations that are possible on these dimensions.

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language Migration*	*Code generation*

Table 1.1: The two orthogonal dimensions of model transformations (copied from [7])

1.2 The problem

Now that the context has been clarified in the previous section, let us focus on the problem at hand. Essentially, what we want to accomplish is to fill the model of an MDE tool with *as much information as possible* from the legacy system. After this, the MDE tool can be leveraged to improve the system, employ new technologies, provide integration with other systems, and so on. In figure 1.1, our goal is depicted schematically. In the lower left corner, the legacy system is shown and is considered to be our source system in the transformation. In the right part of the figure, a model driven system is shown and serves as the target system. Our goal to use MDE to perform the transformation from the source system to the target system, that is represented by the arrow with the question mark. Notice that the endpoint of the arrow is the model in the target implementation. The actual target system will then be generated from the model using the MDE tool at hand. Since the main goal is to modernize (and not necessarily change) the source system, it is important that the target system is similar or even equivalent to the source system. This is depicted with the dashed line between the source system and the target system in figure 1.1.

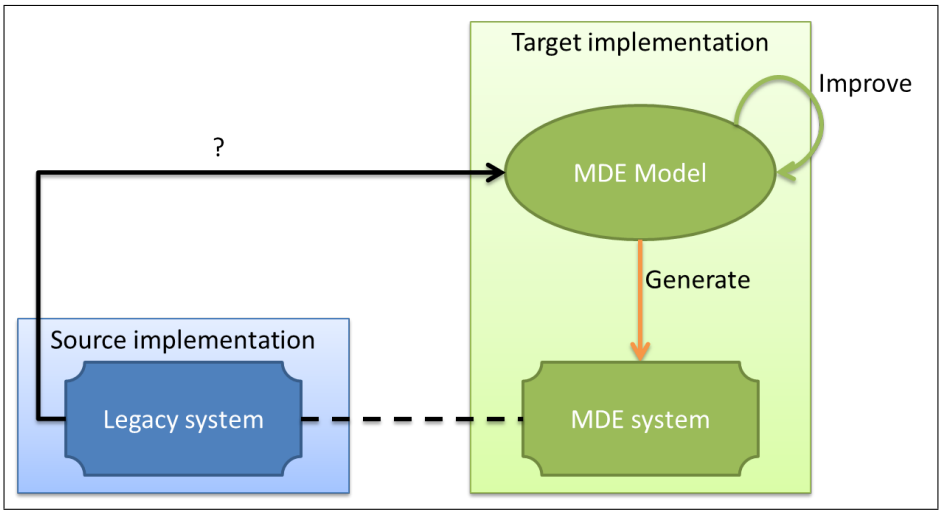

Figure 1.1: Schematic (simplified) overview of the goal.

There are several difficulties that arise when we look at this goal. The first problem is that the available models of legacy systems (if they are even available) are typically out of date, i.e. the systems have been modified and extended, but the models were kept the way they were. This means that a model often has to

be reverse engineered from the source code. Legacy systems also come in a wide variety, i.e. they are heterogeneous. This implies the need for much flexibility in the reverse engineering. Furthermore, we have to transform as much as possible from the source system in order to make the target system *as equivalent as possible* to the source system. This means not only the static structures (such as the data-model) should be transformed, but also the dynamic structures (such as the source code).

The notion of transforming *as much as possible* aims for a best-effort approach instead of all-or-nothing. This means that when something – for whatever reason – cannot be transformed to the target system, we accept this loss at first, create a reminder for this object in the target system, and continue with the parts of the system that *can* be transformed. At a later time, it should be possible to (perhaps manually) transform the remaining parts. This can be achieved by using proxy objects as the "reminders". A proxy object consists of a reference to the actual object in the source system that could not be transformed. In this way, the target system has knowledge about all the objects in the source system, while only a part of these objects might actually be transformed. This prevents the need to analyse the source system again to determine untransformed objects in the future.

Finally, there needs to be some way to measure the equivalence of the target system with the source system. Since solving the equivalence problem for software systems is out of the scope of this thesis, we propose the use of traceability for measuring equivalence. For the traceability, we adopt the idea of "traceability as a model" from [8]. To address these problems, research questions are composed and presented in the next section.

1.3 Research questions

In this section we define the focus of the thesis. As was discussed in the previous section, the main goal is to utilize MDE in the modernization of legacy information systems. We can break this goal down into two steps. The first step is to make the legacy system "model driven", such that an MDE tool can work with it. The second step is to leverage the functionality of the MDE tool to further modernize the system, e.g. create a web interface, integrate with other systems, etcetera. Since the MDE tool is designed for this, the second step will be no different than any other project. Therefore, in this thesis, we focus on the first step; making the legacy system model driven. Ironically however, we try to accomplish this with an MDE approach as well. So, to recap, we focus on *modelling* the transformation from a legacy system to a (equivalent) model driven system.

The main research question is:

> *How can Model Driven Engineering be used in a traceable best-effort method for modernizing legacy systems?*

The main research question can be broken down into the following sub-questions:

- How can a legacy system be modelled effectively?

- What is a tractable meta-model for modelling transformations from the source meta-model to the target meta-model?
- How can traceability be provided automatically?

Requirements on prototype

This project serves two purposes. Next to the scientific contribution for graduating on the University, the prototype will be owned and used in practice by Thinkwise Software [1]. Therefore, there are also requirements on the prototype that have to be taken into account. The following requirements are stated for the prototype:

- The prototype should be easy to use for someone with experience with the Thinkwise Software Factory.
- The prototype should be flexible enough to allow for custom-made code analysis.
- The prototype should be extensible to support new types of legacy information systems in the future.

1.4 Validation

The method is validated with a prototype that is used to transform a sample legacy system. The sample legacy system is a Microsoft Access application. The reason for this is twofold. Firstly, Thinkwise encounters many Access applications at potential clients, which makes it strategically interesting to use Access for the validation. Secondly, Access has a clear structure in which the components of the system are organized. This makes it easier to demonstrate how the transformation is constructed.

For the prototype, the Thinkwise Software Factory, or TSF, is used as the MDE tool. The TSF is extended so that it can be used to create a applications that transform legacy systems into the TSF. These applications are called Upcyclers, and a specific instance is created for Access applications. This makes the TSF not only the enabler for the transformation, but also the target environment of the transformation. The validation will focus on the following points based on the research questions and requirements on the prototype given in section 1.3:

1. Best-effort approach
2. Not only the data-model, but also other models can be transformed
3. Automatic traceability
4. Easy to use for someone with experience with the TSF
5. Allows for the use of custom code
6. Extensible for new legacy information systems

[1] `http://www.thinkwisesoftware.com`

The first point is validated by showing that the target system contains transformed objects, but also untransformed (proxy) objects. Point 2 is validated by showing that, next to the data-model, also parts of the functionality of the Access application is transformed. Point 3 is validated by showing the trace links, that link all objects in the target system to their corresponding objects in the source system, after the transformation is performed. Point 4 is validated by a deduction from the fact that the prototype uses many of the standard screens of the TSF. Furthermore, it also uses the same techniques for building transformations which are also used in other projects. Point 5 is validated by showing the use of custom code for categorizing source code in the Access application. The last point is validated by a deduction from the design of the prototype.

1.5 Contribution

The main contributions of this work are stated below:

- A general validated method for modernizing legacy systems, where:
  - MDE is not only the target environment, but is also used to accomplish the transformation,
  - a best-effort approach is used,
  - not only data-models can be transformed, but also GUI models, Process models, and functionality,
  - automatic traceability is provided.
- A prototype for modernizing Microsoft Access applications into the Thinkwise Software Factory
  - which also serves as a basis for future modernization projects with the TSF.

1.6 Overview

The thesis is further organized as follows. In the next chapter (chapter 2), related work and the state of the art in the field of software modernization is discussed. In chapter 3, our general approach to the main research question is presented. The first sub-question will be addressed in chapter 4, and the second and third in chapter 5. In the chapter thereafter (chapter 6), we give a more detailed introduction to the Thinkwise Software Factory. This is important, because the TSF was used to develop the prototype for the validation. After this, in chapter 7, we discuss the validation of the method. We finish the thesis with a detailed discussion about the results and also elaborate on interesting opportunities for future work.

Chapter 2

Related Work

In this chapter, related approaches to dealing with legacy systems and model transformations are discussed. The first section is about available literature on the replacement of legacy systems in general. From this work we can construct a general strategy on how to deal with legacy systems. The section thereafter deals with other model driven approaches to the modernization of legacy systems.

2.1 Dealing with legacy systems

In this section, literature is discussed regarding the migration of legacy systems in general. The migration of legacy systems is not a new problem, so therefore, we first explore the existing strategies for dealing with legacy systems.

2.1.1 Transformation strategies

In [1], Ulrich provides a comprehensive set of guidelines for the transformation of legacy systems. The migration is approached from a business perspective as well as from a technological perspective. Guidelines are provided for making a business case, and the difficulties and common pitfalls are addressed. Furthermore, a survey of migrating methodologies, accompanied with techniques, is provided. The book provides a clear overview on the difficult task of migrating legacy systems. The author based the work on his own experience with legacy systems. The principles in the book inspired our ideas for the general approach discussed in chapter 3.

2.1.2 Deployment strategies

Brodie et al. present the Chicken Little approach in [9]. The approach is mostly concerned about the deployment of a migration of a legacy system. From practical experience, the author found that the migration strategies at that time were not adequate for large systems. The Chicken Little approach

gets its name from performing the migration using little steps at the time. This is realized by creating gateways in-between the components of the legacy system. A particular component can then be redeveloped, and tested while the production environment still uses the original component. When the new component is tested and accepted, the gateway can be switched over, to relocate the traffic from the other components towards the new component. This allows for stepwise migration in situations where the legacy system cannot be taken off-line. The book mainly focusses on the issues of deployment, and provides a validated treatment for those problems. The book does not elaborate on how the migrated components are created, whereas in this work, this is our main focus. Therefore, we did not use this directly in our approach. However, because it can be very useful in the deployment of the new system, it is good to mention it anyway, perhaps for future research.

The previous subsections explored existing approaches to the migration of legacy systems. Our goal is to accomplish this using Model Driven Engineering. Therefore, in the following section, we discuss the most prominent approaches towards MDE.

2.2 Model Driven Architecture

The Model Driven Architecture (or MDA [6]) is the vision of the Object Management Group (OMG) [10] on Model Driven Engineering. The key principle is that the platform on which a system should run is abstracted away in the development of the system. A Platform Independent Model, or PIM is used for this. Then for the platform(s) that should be used, a Platform Specific Model, or PSM, can be generated automatically from the PIM. This allows for a system to switch platforms more easily, and lets the developer concentrate more on the logic of the actual system instead of the platform.

2.2.1 Meta-Object Facility

The Meta-Object Facility, or MOF [11], is composed of four levels of models and meta-models. See figure 2.1.

The lowest level, m_0, are for real world objects. The level above that, m_1, models these objects. In the level above that, m_2, the meta-model of the model in m_1 is positioned. Recall that a meta-model defines the abstract syntax of a modelling language. In this case, this is the language in which m_1 is expressed. An example of such a meta-model is UML, but there can also be other languages defined by a meta-model at m_2. The uppermost level, m_3, is for the meta-meta-model, which in MDA is MOF. MOF is essentially a subset of the UML class diagram. Just like the UML in m_2 is a meta-model for the model in m_1, the MOF in m_3 is a meta-model for UML. The MOF is the highest level, since it is also a meta-model for itself. This means that all the constructs in MOF can be modelled using only those constructs. In general, there is a *conforms to* relationship from a model in m_n to the corresponding model in m_{n+1}.

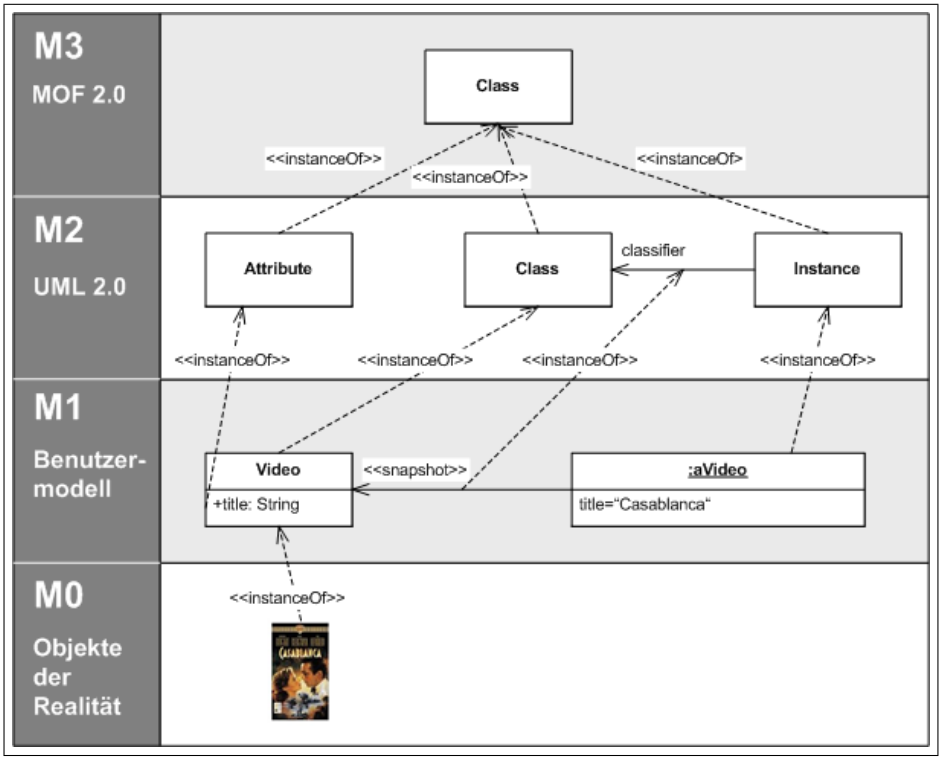

Figure 2.1: The four layered architecture of the Meta Object Facility (from [12])

2.2.2 ATL

ATL is short for ATLAS Transformation Language. It is a model transformation language based on MDA [13]. ATL is a hybrid language in the sense that it supports declarative as well as imperative definitions of transformation rules. This makes it a powerful language which provides much freedom for the developer. The declarative notation is preferred, since that is the cleanest way to define a transformation and also allows for better control. The imperative notation can be used when a declarative notation would become too difficult to construct. In this way, the benefits of both paradigms are utilized and can be put into practice where appropriate. We appreciate this approach, but we prefer to use a model at a higher abstraction level to define the transformation. This allows us to benefit from the existing MDE techniques when defining and executing the transformation. In chapter 5 this is made more clear.

2.3 Reverse engineering

Since the legacy system itself is often the most important resource in the migration project, in this section, we discuss existing techniques for reverse engineering.

2.3.1 Database reverse engineering

In [14] several techniques are discussed to reverse engineer databases. A distinction is made between explicit constructs and implicit constructs, which we find useful. Explicit constructs are things like primary keys, that are created using the Data Definition Language (DDL) of the (Relational) Database Management System (RDBMS). These constructs often can be queried, and therefore are trivial to reverse engineer. The implicit constructs, however, are more difficult to reverse engineer. Implicit constructs – as the name implies – are present in a system, but can not be queried directly. So their presence needs to be derived. For this, data or code can be used. An example of an implicit construct is a foreign key, that is not defined explicitly as such. Another example would be a validation in a trigger.

2.3.2 Model discovery

The MoDisco project is an Eclipse based project for the reverse engineering of models from software systems [15]. The project is based on ECore, which is the meta-meta-model for the Eclipse Modelling Framework, EMF [16]. ECore is a simplification of OMG's MOF, which are both m_3 level languages. The goal is to create a standard platform in which models can be described and derived from existing applications. When a model is created, techniques that are already available on ECore can then be used to transform or extend the application. At the time of writing, the project was able to reverse engineer Java code, but we could not yet use it for our method.

2.4 Transformation approaches

Let us assume, that with the reverse engineering techniques from the previous section, a model can be constructed for the legacy system. Then it is now important to transform this model so that it can be used with the MDE tool of choice. This process is referred to as model transformation. In our quest to find solutions to this problem, we found interesting literature from the model domain as well as from the data domain.

2.4.1 Architecture driven modernization

The Object Management Group designed the so called Architecture driven modernization approach [17]. It is based on MDA, but in a sense, reversed. It is no

coincidence that the abbreviation ADM, is MDA in reverse. A somewhat simplified version of the approach is as follows. First, a model from a legacy system is created. This is what is called a platform specific model of the application, or PSM. This is a term from the MDA. The next step is to transform this model into a platform independent model, or PIM. The PIM is, by definition, a more abstract version of the platform specific model. This step is what is meant by MDA in reverse. In normal MDA, first a PIM is created, from which a PSM is derived / generated. Now, first a PSM is available, from which a PIM is created. The idea is, that once a PIM is available from the application, the rest of the project is just as any other MDA project. This means that the transformation from the PIM to the target PSM, can be carried out with existing tooling. In the next section, we discuss a somewhat comparable approach to the PIM, found in the database domain.

To accomplish this method, OMG introduced the Knowledge Discovery Meta-model (KDM) [18, 19]. This is a comprehensive meta-model to store information about existing systems. We can characterize the ADM approach as a bottom up approach, since we start with a very specific model, and then work towards a more abstract version. KDM is very large and contains many abstractions. The abstractions mean there are different implementations possible for a specific construct. In effect, this means that the KDM actually is a set of possible meta-models. This refrained us from using it, since we prefer the simplicity of a specific fixed meta-model for every type of legacy system. This is described in more detail in chapter 4.

2.4.2 ModelGen

An interesting approach to schema transformation is the work of Atzeni et al. [20, 21] on ModelGen. ModelGen is "the model management operator to translate schemes from one model to another." In this context, a *model* is the language in which a data scheme can be described, for instance: relational, XSD (XML schema), or object oriented. A scheme is then a specific description of how data should be stored, using the constructs provided by the model. For example, in an object oriented model, there are the constructs `Class` and `Field`, which can be used in an object oriented scheme for an employee class and name field respectively. The main contribution is the introduction of a supermodel which generalizes all other models, and is described by the so called: meta-supermodel. The supermodel has constructs like `Abstract` and `AttributeOfAbstract`, which correspond to the constructs `Class` and `Field` from the object oriented model respectively. For each model that needs to be supported, specific meta-models can be created, but the meta-models are always a subset of the meta-supermodel. That is, all the constructs (and their properties) used in a model specific meta-model have a counterpart in the meta-supermodel, and are directly connected to it. The constructs `Class` (object oriented), `Entity` (Relational) and `Node` (XSD) for instance, are all connected to the same `Abstract` construct in the supermodel, and all share the `Name` property. The supermodel is a model independent representation of the data scheme, much like the PIM is platform independent representation of a model in ADM. The meta-models can be instantiated into models, in which then the

data schemes can be stored.

A consequence of the fact that all meta-models are subsets of the meta-supermodel, is that the expressiveness in the models is constrained to the expressiveness of the supermodel. The author claims, however, that the constructs in the supermodel are indeed sufficient to express a wide variety of (data) models. The main benefit of this approach is that for each model only the translation from and to the supermodel should be specified (in terms of the meta-constructs) to be able to translate a scheme from any model to any other model. For instance, for the relational model, it should be specified how the properties of the `Entity` construct need to be translated to the properties of the `Abstract` construct in the supermodel and vice versa. When this is also done for the construct `Class` in the object oriented model, we are able to translate instantiations of these constructs from the relational model to the object oriented model and vice versa – using the supermodel model as a mediator. Again, notice the similarities with ADM in the previous section. In the traditional approach, where a translation is required for every pair of models, we would need n^2 translations, whereas with this approach, only $2n$ translations are required for n models.

While this also seems like an elegant approach for the exogenous model transformations in which we are interested, there are some limitations that prevented us from applying it. First of all, we would need to extend the supermodel such that not only models (or schemes) in the data domain can be expressed, but models of complete information systems. Models of information systems typically do *contain* a data model, but also include GUI, process and functionality models. Because there is such a wide variety in how these concepts are implemented (in particular the functionality), it is infeasible to compose a clear supermodel which generalizes all these implementations. Recall that the KDM approaches this, but only with extensive use of inheritance hierarchies. However, there is another limitation that makes this approach inappropriate for our application. This has to do with the fact that the constructs in every meta-model are directly connected to the constructs in the meta-supermodel. This would allow only for very straightforward one to one transformations (on the connected constructs), where more elaborate transformations are required when transforming models of complete information systems. Finally, the greatest benefit of the approach, that only $2n$ translations are required for n models, is less significant in our application since our target (meta-)model typically is fixed.

2.5 Conclusion

In this chapter we have explored the state of the art of the most important aspect of our own work. The principles from Ulrich [1], served as a good starting point in the development of our general approach found in the next chapter. Furthermore, we gave pointers to techniques for reverse engineering. Since this work focusses more on the overall approach of migrating legacy systems with MDE, we did not employ very elaborate reverse engineering techniques. Future research should point out how the techniques in [14] and [15] and others can be applied effectively in our approach. Since our goal is to modernize legacy systems with the use of MDE, we also looked at the relevant standards devel-

oped by the OMG; the Architecture Driven Modernization and the Knowledge Discovery Meta-model. Finally, we drew a parallel between the Platform Independent Model in ADM to Atzeni's supermodel in his work on ModelGen [21]. In both cases, an intermediate level is created which generalizes all "environments" (in the former case: platforms, in the latter case: models), to aid in the transformation from one environment into the other. We do not believe that this will be feasible with entire legacy systems, and we therefore did not employ such an approach. The approach that we did use is discussed in the following chapter.

Chapter 3

General approach

In this chapter we present our approach on transforming legacy systems with MDE. The main issues involved have been identified already in the introduction. The focus is on how to create a model from a legacy system that can be used by a specific MDE tool. We assume that the MDE tool uses a fixed meta-model to which the all the models have to conform to. Only when a model conforms to the meta-model, the tool can be used to extend and modify the system. In this way, the meta-model of the tool acts as a constraint on the models that can be used. Recall figure 1.1 in which the goal is represented by the arrow indicated with a question mark. The arrow represents the creation of a model from the legacy system that conforms to the meta-model of the MDE tool. If we look closely, there are actually two main problems in this process.

The first problem is that the MDE tool and the legacy system typically use a different technology to store their information in. The MDE tool can, for instance, store its models using XML or a relational database, while the legacy system could use COBOL or Access files. We refer to this concept as the *technical space* [22] in which the information resides, and is discussed in more detail in section 3.1. The problem is that this difference in technology needs to be bridged. That is, the *information* in the technical space of the legacy system needs to be extracted and put in the technical space that the MDE tool uses.

The second main problem of the general approach is that the information about the legacy system is expressed using concepts that do not (all) exist in the MDE tool. Consider, for example, a legacy system that uses a textual menu system as the user interface, but the MDE tool only works with graphical user interfaces. The information about the menu system cannot be put directly into the MDE tool, since there is no place for it. This problem occurs because the (implied) model of the legacy system has a different meta-model than the meta-model of the MDE tool.

Since the above two problems are conceptually different, it is wise to treat them separately. In figure 3.1, this division is shown. In the first step, a model of the legacy system is created. This step mainly involves the use of custom code to reverse engineer the legacy system and output a model. This model serves as the reference of the legacy system (for instance in the traceability), and is

the starting point of the transformation. The second step is to transform this model in such a way that it conforms to the meta-model of the MDE tool in question. Since the first step already has been done at this point, the input in the second step is always a model. It does not yet have the meta-model of the MDE tool, but it is still a model. When this model is stored using the same technology (or technical space) as is used by the MDE tool, we can leverage the MDE tool to accomplish the transformation. In this way, the transformation is defined as a model that expresses how models with the meta-model of the legacy system should be transformed into models with the meta-model of the MDE tool. Just like with models for software systems, a model for such a transformation provides a higher abstraction level. Furthermore, much of the other benefits of MDE can also be applied to transformation models. An example is a graphical visualization of a transformation model, or a validation to detect conflicting transformation rules. Another important benefit in our case is that the MDE tool can also be used for the automatic traceability.

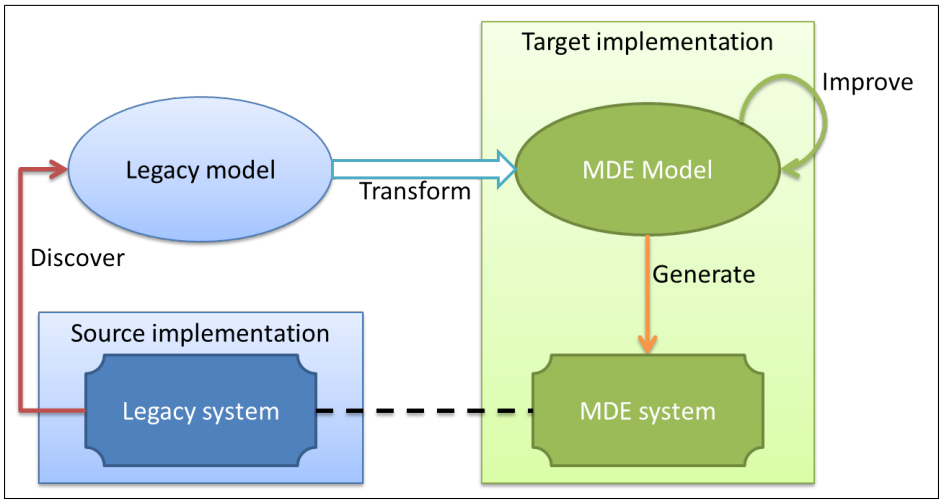

Figure 3.1: General approach broken down into two separate steps.

3.1 Technical spaces

A technical space [22] represents a certain technology accompanied with tooling. It can be seen as a carrier in which information can be stored and modified. A simple example of a technical space is a piece of paper. The technology is then paper, and the accompanied tools are, for example: a pencil, scissors and a copier. In our technical space, we can now add information using the pencil, remove information with the scissors and copy information with the copier. The paper, however, does not prescribe a language that needs to be used to express the information in. It supports all languages that can be written with a pencil. This means it is not a meta-model, but a meta-meta-model. The language that we choose to write in, serves as our meta-model – English, for example. The English text which will then be written on the paper, is the model.

3.2 Exogenous model transformation with MDE

The notion that MDE can be used for exogenous model transformations (model transformations where source and target meta-models are different) deserves some more attention. The way we approach this, is to first consider a transformation as something that we can create a model for. This seems straightforward, but before we continue, let us examine what is needed in order to do this. See figure 3.2. To define a model for an exogenous model transformation, one needs to be able to express what constructs in the source meta-model correspond to what constructs in the target meta-model. This means that the transformation model is at the level of the meta-models of the source and target system (instead of at the model level). The benefit of defining the transformation at the meta-model level, is that a given transformation can be reused on multiple source models which share the same meta-model.

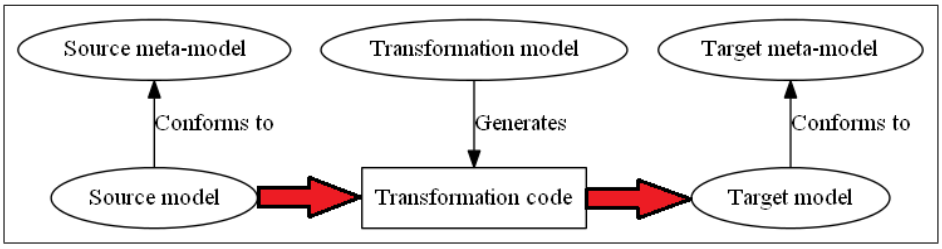

Figure 3.2: Positioning of the transformation model

Furthermore, to define the correspondences between the source and target meta-model, we need both meta-models to be at hand. Or more specifically, both the meta-model of the source model and the meta-model of the target model need to be stored explicitly and be accessible to the MDE tool. In the next chapter we see that explicitly modelling the meta-model of the source system is part of our method. So this meta-model will be available. Recall that the target meta-model is the meta-model of the MDE tool used. This means that we need the meta-model of the MDE tool to be accessible to the tool itself – it should be able to read its own meta-model. The MDE tool we use in the validation, the Thinkwise Software Factory, is able to do this. Finally, the MDE tool also needs to be extended to be able to create and read transformation models. For this, a meta-model for model transformations is designed and incorporated in the meta-model of the MDE tool. In chapter 5, we discuss the design of our meta-model for model transformations.

Once a model for a transformation is created, we then use it – in conformity with MDE – to generate the code which actually performs the transformation. This entails a further extension of the MDE tool, to not only be able to read transformation models, but also generate executable transformation code from them to perform the transformation. Because the source model, the target model, and the MDE tool are already in the same technical space, the transformation can also be performed in that same technical space.

3.3 Metaphorical example

We can use the example from section 3.1 as a metaphor to illustrate the different steps in our general approach. The information in a legacy system is comparable to the English text written on paper. This is the source model. The target model is Dutch text in a digital TXT file. The strategy that we employ, would be to first scan the paper with the English text into a TXT file. This is a change in technical space, since the information now resides in a TXT file. However, it is still in English, so this resembles the first step in our general approach. The second step is to translate the English text into Dutch text. The English language is comparable to the meta-model of the legacy system, and the Dutch language is comparable to the meta-model of the MDE tool. The transformation is defined by mapping the English words onto the corresponding Dutch words, much like a dictionary. This "mapping" resembles the transformation model in our approach. To execute the transformation, the English text is parsed, and converted word by word to produce a TXT file with the Dutch text. In our approach this transformation logic is generated from the transformation model – in conformity with MDE.

The example illustrates the general approach which begins to answer our main research question. Recall the main research question: How can Model Driven Engineering be used in a traceable and best-effort method for modernizing legacy systems? From the example, we can see where MDE can be used in the approach – this takes care of the first part of the research question. We did not yet discuss the aspects in the question about traceability and best-effort. However, we can incorporate these easily in our example as well. Let us first look at traceability. The traceability aspect requires that objects from the target model, can be traced back to objects from the source model. In the case of our metaphorical example, this would mean a trace from the Dutch words in the translated text to the English words in the original text. The creation of these links, would occur automatically when the translation is executed. The best-effort aspect can also be incorporated in our example. The best-effort principle means that when something cannot be transformed, we do not abort the transformation, but instead continue with the parts that can be transformed. Examples of English words that cannot be translated directly into Dutch are: cool, okay, or proverbs. These parts are often just copied as-is, into the Dutch text (possibly with quotes around them), or manually described in different words.

3.4 Conclusion

In this chapter, we have outlined our general approach that serves as the main strategy throughout the rest of the thesis. The approach makes an important distinction between the different steps that need to be undertaken in transforming legacy systems into model driven systems. We proposed to first perform a transformation in technical space, before a transformation in meta-model takes place. Since the first step places the source model in the same technical space as the MDE tool, the second step can be performed in that technical space as well,

using the MDE tool. A model is created for the transformation, from which the executable transformation code is generated. This reduces the amount of effort needed for the developer to define and perform the transformation.

We have shown the traceability and best-effort properties of the general approach, and that it is a superficial answer to the main research question. Obviously, a complete answer to the main research question requires more details about how exactly the different steps should be implemented. This is done by answering the sub-questions in the following chapters. In the next chapter we answer the first sub-question about how exactly we should model legacy systems. The chapter thereafter answers the second and third sub-question, about how transformations should be modelled and how automatic traceability can be provided, respectively.

Chapter 4

Modelling a legacy system

In this chapter we focus on the first step in the general approach presented in the previous chapter. This first step was to move the information in the legacy system into the technical space of the MDE tool. In our case, this roughly means creating a model from the legacy system and storing it in the way the MDE tool does this. While this fixes the technical space in which the model is created, there are several strategies possible regarding the meta-model that should be used. This chapter explores these possibilities, and provides a motivation for the choice we made. We thereby answer the first sub-question: How can a legacy system be modelled effectively?

The first section gives a general introduction to the relation between data and meta-data. In the section thereafter, the theory is applied to models and meta-models. In section 4.3, we compare the benefits and drawbacks of generic and specific meta-models, and we explain our preference in this regard. Lastly, section 4.4 discusses OMG's Knowledge Discovery Meta-model, and we conclude the chapter in section 5.7.

4.1 Different ways to express information

In this section, we give a general introduction to the relationship between data and meta-data. This will help us in the following sections to reason about the benefits and drawbacks of generic and specific meta-models.

To store information, we generally encode it in data parts and meta-data parts. The meta-data defines the structure, and the data provides the instances of the structure. To retrieve the information, both the data *and* the meta-data is needed. To illustrate this, take a look at table 4.1.

We can see part of the information, but we do not know what the data means. Therefore, unless we have the meta-data, it is completely useless. In table 4.2, the meta-data is added, and suddenly everything makes sense. The mysterious numbers from table 4.1 have now been given meaning. In the opposite case, where only the meta-data is known without the data, we also do not know

?		
?	?	?
516874161	1995	Yes
156489465	1998	No

Table 4.1: Data without meta-data

Alumni		
Social Security Number	Year of graduation	With honours
516874161	1995	Yes
156489465	1998	No

Table 4.2: Data with meta-data

what the information is. The meta-data is only a structure, and until it is instantiated, we do not know the specifics. However, in this case, we do know something *about* the data. If you look at table 4.2, and cover the lower (data) part of it, we do know that alumni are stored in this table. Furthermore, we know that they *have* a social security number, that there *is* a date of graduation, and that the alumnus can be either graduated *with honours* or *not*.

In general, we can define the parts of information in the data as I_D and the parts of information in the meta-data as I_M. The *instance of* relation r between the data parts and the corresponding meta-data parts is denoted as:

$$r : I_D \rightarrow I_M$$

In the previous example, r would link the data part: 1995 to the meta-data part: Year of graduation.

With these definitions, we can also define the total amount of information I as a combination of the parts of information in the data I_D and in the meta-data I_M as:

$$I = I_D \bowtie_r I_M$$

From this definition we can infer that in situations where the total amount of information I remains equal:

- when the amount of information in the meta-data I_M gets larger, the amount of information in the data I_D gets smaller, and
- when the amount of information in the meta-data I_M gets smaller, the amount of information in the data I_D gets larger.

We shall illustrate the second statement with an example. The first statement can be inferred directly from this example as well. Consider the meta-data in table 4.3. This is an example of very generic meta-data. If we compare this to table 4.2, we see that the amount of information in the meta-data, or I_M is lower in this case. When there is no data, we only know that there can be objects, and that these objects can have properties, with values. According to the second statement above, to capture the same amount of information about the alumni with this meta-data, the information in the data gets larger. This is shown in table 4.4.

Object	
id	type

Object_property		
object_id	prop_id	value

Table 4.3: Generic meta-data

Object	
id	type
1	Alumnus
2	Alumnus

Object_property		
object_id	prop_id	value
1	Social security Number	516874161
1	Year of graduation	1995
1	With honours	Yes
2	Social security Number	156489465
2	Year of graduation	1998
2	With honours	No

Table 4.4: Generic meta-data with data

Compare the amount of tuples in table 4.4 and table 4.2. Not only are there 6 more tuples in the generic structure, the tuples also contain more information. The tuples in object_property for instance, not only contain the value of the property, but also the type of property. In table 4.2, this kind of information was stored in the meta-data instead of in the data.

4.2 Models and meta-models

The relation between data and meta-data discussed in the previous section also applies to models and meta-models. A model is the data, and the meta-model is the meta-data. From the previous section we have seen, that this means we have a choice in how much information we put in a meta-model, and that this choice affects the amount of information in the model. Furthermore, the information that is stored in the meta-model is fixed, i.e. every model will have the same structure imposed by its meta-model. The information stored in the model is variable – this will change with each model.

The fact that the information in the meta-model is fixed, can be exploited by MDE tools for providing "model-generic" functionality. Consider, for example, a meta-model in which a relational data scheme can be modelled. When the meta-model exposes the fact that there are tables and columns and prescribes what properties these entities can have, we know that every model will use this structure to model relational schemes. This knowledge *about* the models, can be used to write code that generates DDL statements to create the actual tables in an RDBMS. This code is model generic, because it will work with *any* model that conforms to this meta-model. When the meta-model would only contain very generic constructs, like "objects", and "properties of objects", this would not be possible since an "object" can be anything.

The MDE tools that we focus on, such as the Thinkwise Software Factory, leverage the principle of putting more information in the meta-model to provide

model-generic functionality. As a consequence, we see in the next chapter that the use of such a specific meta-model also imposes many constraints on the target model that need to be accounted for in the transformation. In the next section we examine if the meta-model for the source model should be generic or specific.

4.3 Generic or specific meta-model

Since we need to capture the information in the legacy system in a model, we need to decide what meta-model we should use. There are several approaches possible. Because there is a wide variety of legacy systems, we might be inclined to use a very generic meta-model – that is, a meta-model with little information about the system. That way, we have a lot of freedom in the model, because there are few restriction on what the model should look like. However, solely modelling the legacy system is not the only purpose here. We also need to be able to transform this model into a model which conforms to the meta-model of the MDE tool.

In chapter 3, we stated that a transformation model is created with the correspondences between the source meta-model and the target meta-model. This means that the meta-model that we choose for the source model has a great impact on the effectiveness of the transformation model. As we saw in the previous section, the use of model-generic functionality diminishes when the information in the meta-model gets smaller. In this case, model-generic functionality translates to transformation logic. This means that the use of a very generic meta-model – to generalize all legacy systems – would not work well in our approach, since we cannot define useful transformations on such a meta-model.

If, on the other hand, we make the source meta-model too specific, we run the risk that only few legacy systems can be modelled with it. In that case, we can define a good transformation model for it. However, since this meta-model – and consequently the transformation model – can only be used for a few specific legacy systems, this is not very model-generic either. So choosing the right meta-model is a trade-off between generalizing legacy systems and the ability to use useful model-generic functionality.

Our recommendation to modelling legacy systems would be to use specific meta-models which generalize a certain *type* of systems. Examples of types of systems are: COBOL applications, Access applications, Lotus Notes applications, etcetera. In this way, we can use the meta-model for more than one legacy system while we also can employ model-generic functionality.

4.4 Knowledge Discovery Meta-model

While we do not recommend using a very generic meta-model because it limits the use of model-generic code, there is an exception to this notion. OMG's

Knowledge Discovery Meta-model, or KDM, is a standard meta-model for information about existing software. Since it is a standard, it could be the case that there is already software available to create a KDM model from a specific legacy system. It would be a waste if that could not be used in our approach. While this meta-model contains many general and abstract constructs, it also contains possible specific instantiations of these abstract constructs. Using these more specific constructs it is possible to write model-generic code for the KDM. Therefore, our approach is, in principle, compatible with the KDM. However, there is also a downside in using the KDM for the source model. This concerns the fact that in order to fit the information from a legacy system in the KDM, an implicit model transformation occurs (from the legacy system into the KDM). This potentially causes a greater loss of information than when a custom, more specific, meta-model is used. This loss will not be detected with the automatic traceability, since that only traces the model transformation that occurs after this first step.

4.5 Conclusion

In this chapter we discussed various ways to express information in meta-data and data. We have seen that using a very generic meta-model is not useful, since it prevents the use of model-generic functionality. The best way is to generalize only a certain type legacy system in a meta-model. This allows for reuse of the meta-model and contains enough information to use model-generic functionality for the transformation. Thereby, this chapter answered the first sub-question. This result is used in the next chapter, where we continue with the second and third sub-question about the transformation model and automatic traceability.

Chapter 5

Modelling model transformations

In this chapter we focus on the second and third sub-questions. The second sub-question is: What is a tractable meta-model for modelling transformations from the source meta-model to the target meta-model? The third sub-question is: How can traceability be provided automatically? These sub-questions both concern the second step in the general approach, the model transformation. At this point, the legacy system is already represented by a model in the technical space of the MDE tool, but its meta-model is different from that of the tool. The result of the transformation should be a model that does conform to the meta-model of the tool. In this chapter we design a meta-model for model transformations, and show how automatic traceability can be provided.

In the next section, we first discuss the reasons for using a model for the transformation. After this, we discuss the various requirements on the transformation that need to be taken into account. We continue by reducing the model transformation problem to that of schema mapping in section 5.3. In the section thereafter, we discuss the cardinality of entities and the use of pre-processors. In section 5.5, we discuss two detailed examples of schema mappings, to make clear what needs to stored in the meta-model. After this, in section 5.6, we present the transformation meta-model. We conclude the chapter in section 5.7.

5.1 Why model the transformation?

To transform models, many transformation languages have been developed [23]. Instead of creating another transformation language, our approach is to *model* the transformation and generate the transformation code from it. The benefits of modelling transformations over programming them (i.e. using a transformation language) is that models are more synoptic and can therefore be created and maintained easier. And if the same MDE tool is used, modelling the transformation is not very different from modelling models that we already know: no

extra language needs to be learned to create transformations. Furthermore, if the MDE tool already has facilities for code generation, this can also be reused for the transformation model. Another benefit of using a model, is the fact that models are easy to analyse. We could, for example, analyse if a transformation contains conflicting rules. Since the MDE tools are already built to perform these kinds of analyses, it would be easy to provide this also for the transformation model. With a new transformation language, this would be more difficult, since the transformation definition then needs to be parsed and interpreted first.

Transformation rules

It is common practice to divide a model transformation into smaller steps: transformation rules. This can be seen in many other approaches including ATL [13]. A transformation rule is a logical sub part of a complete transformation. A transformation rule specifies how a certain group of objects in the source model should be transformed into what objects in the target model. The whole transformation is comprised by the complete set of the transformation rules.

Between transformation rules there can exist dependencies. A dependency between two transformation rules, means that the dependant rule can only be executed after the other rule is executed first. As such, the dependency causes an ordering to be formed in the execution of the transformation rules. Furthermore, two transformation rules could conflict with each other, when they are mutually exclusive.

5.2 Requirements

From the research questions stated in chapter 1, we deduced several requirements regarding the transformation and the transformation meta-model. These requirements are shown below:

- The transformation models should be executable
- The transformation meta-model is tractable
- Automatic traceability can be provided
- Custom analysis can be performed

The first and most important requirement is that a transformation model should *completely* define a transformation from the source model to the target model such that it is executable. This means that the model deterministically defines what actions need to be performed to realize the transformation. The second requirement is taken directly from the second research sub-question, and requires that the transformation meta-model is tractable. Since “tractable” is a subjective quality, it makes it hard to quantify it. We mainly want to prevent that the transformation meta-model gets overly complex as a consequence of supporting some rare transformation scenarios. The next requirement is represented by the third sub-question, and requires that automatic traceability can be provided. This requirement is more concerned on the execution of the transformation,

rather than on the (meta-)model of the transformation. It means, that when the transformation is executed, trace links are automatically created to link the source object with the target object and the corresponding transformation rule. The last requirement is that custom code can be used for more elaborate transformations that are not natively supported by the meta-model. An example of a more elaborate task would be the extraction of implicit constructs [14] in the source model.

5.3 Schema mapping

To make more clear what our goal is with a transformation model, we first look closer at the problem we need to solve. Our objective is to transform a model in such a way that it conforms to the target meta-model. This is quite an abstract definition, which makes it hard to specify the actions that need to be done to accomplish it. However, we can reduce this problem to a more specific one, that is arguably more comprehensible. In order to do this, we use the same correspondences between (meta-)data and (meta-)models as we have seen in chapter 4. This means that, a meta-model will translate into a relational data scheme, and a model will translate to the data in that scheme.

The objective to transform a model such that it conforms to the target meta-model, reduces then to transforming the source *data* such that it fits in the target *scheme*. In database research, this is accomplished with the use of schema mapping. Because we represented models and meta-models by data and schemes, we now reduced the model transformation problem to a schema mapping problem. In the following, we therefore actually present a method for schema mapping. Just remember that this method is an implementation of the model transformation in our general approach.

Schema elements

Schema mapping comes down to defining which elements of the target scheme correspond to (or can be found in) the source scheme. So what elements are there? The elements on which we focus from a relational data scheme are:

- Entities
  - Tables with primary keys
- Attributes
  - Columns
- Relations
  - Foreign keys

Let us first consider entities. In a relational database, an entity is represented by a table. We can differentiate entities into two categories: strong entities and weak entities. A strong entity exists on its own, independent of other entities. A weak entity cannot exist on its own: its existence is dependent on another

entity. An example of a strong entity is a person, and an example of a weak entity is a marriage. A marriage is weak because it can only exist when the two persons that got married also exist. When mapping entities from the source scheme to the target scheme, we therefore need to start with a strong entity.

Attributes contain the values that belong to an entity. In a relational database, an attribute is represented by a column. An attribute can only be mapped when the entity that it belongs to is already mapped. So, the attribute mapping depends on its entity mapping.

Relations in this context are *connections* between entities (instead of entities themselves). A relation between two entities is represented in a relational database by a foreign key constraint. Relations play a major role in our schema mapping approach, because they contain much information about the overall structure of a scheme.

5.4 Cardinality

We identify two important requirements on a mapping between two entities. The first is that the two entities must correspond to each other semantically. This means that they represent the same, or a similar concept in both systems. The second requirement constricts this further, by requiring that the entities have the same cardinality. This means that the number of occurrences of the concept in the source system is equal to the number of occurrences of the corresponding concept in the target system, i.e. they relate one to one. Our transformation meta-model can only map entities that conform to these requirements. If the cardinality is different between two similar entities, the source scheme needs to be pre-processed to enable the use of the transformation model. The cardinality between two entities can be different if, for instance, the source system has a denormalized entity, which is normalized in the target system. To solve the issue, we need to apply a grouping operation on the denormalized entity, and store the result in a new entity. The new entity can then be mapped to the entity in the target scheme. The grouping operation is what we call a "pre-processor", and this is discussed further in the next subsection.

Pre-processors

In order to keep the transformation meta-model tractable, we opted for the use of custom pre-processors in cases where one to one mappings do not suffice. Having a simple transformation model makes it easier to create it, analyse it and process it. We expect that in most cases the transformation model would be sufficient. Only when "heavy lifting" like grouping, splitting, or even data-mining needs be performed, a pre-processor is needed. In this way, the transformation model stays relatively clean, and the pre-processors can tackle the more complex issues.

Figure 5.1 illustrates a schematic example of the use of a pre-processor. In the figure, we can see that entity A in the source model is mapped directly onto entity D in the target model. This means that entities A and D are similar entities, and they have the same cardinality. The dashed line between entity B

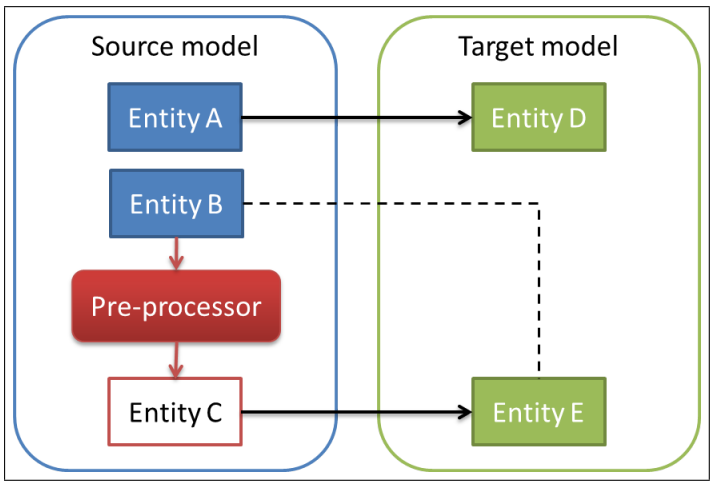

Figure 5.1: A schematic example of entity mappings where a pre-processor is used for entity B.

in the source model and entity E in the target model means that these entities are also similar. In this case, however, the entities cannot be mapped directly. This can have several reasons. The entities could, for instance, not have the same cardinality, i.e. they do not correspond one-to-one. Another possibility is that the information in entity B requires some complex analysis, like parsing or data mining, to extract the relevant parts. Therefore, instead of directly mapping entity B to entity E, the source model is extended with an extra entity: entity C, which is the result of a pre-processor for entity B. This new entity C does have the same cardinality of entity E, and can therefore be mapped to entity E.

A pre-processor can be seen as a custom function that takes an entity from the source scheme as input and produces output in a new entity in the source scheme. The output is then used in the transformation model. Therefore, the pre-processor is also part of the transformation, which means that trace links need to be created for its actions. Since the pre-processor contains custom code, this is not trivial.

However, when a pre-processor can do its work with only one tuple at the time, a framework can be constructed in which the pre-processor is called for each tuple in the input entity. The framework then always knows what the current input tuple is, and also receives what the pre-processor outputs. The framework is then able to create the appropriate trace links for the pre-processor. The trace links that are created for a pre-processor consist of a combination of the input tuple, the output tuple, and an identifier for the pre-processor. The input and output entity of a pre-processor always reside in the source model. A framework like this is also used in chapter 7.

Automatic traceability is harder to realize for set-based pre-processors. These pre-processors cannot be put in the framework mentioned above. In this case it is probably easier to manually create the trace links in the pre-processor itself. We did not focus on this type of pre-processor in this work. Perhaps, in future research, a framework to provide automatic traceability can be constructed that

also supports set-based pre-processors.

Since both the input and output entity of a pre-processor reside in the source model, it is also possible to "chain" multiple pre-processors. The output entity of one pre-processor is then used as the input entity in the next pre-processor. The output of this second pre-processor can, in turn, be the input for a third pre-processor, and so on. It is obvious that the execution of these pre-processors requires an ordering. The ordering is determined by the chain; a pre-processor can only be executed when its input entity is not an output entity of another pre-processor, or when this other pre-processor is already executed. The trace links for these pre-processors also form a chain. Furthermore, the trace links are transitive, such that: $a \rightarrow b \rightarrow c$ results in: $a \rightarrow c$, where $a, b, c \in Tuples$ and $\rightarrow$ denotes a trace link.

5.5 Example mappings

Before we present the transformation meta-model, let us first consider two examples of the kinds of transformation models (mappings) that need to be supported by it. This will help in clarifying the problems and the design decisions for the meta-model. The examples are based on two different meta-models (A and B) for modelling forms in a GUI. In figure 5.2, the different forms are shown that can be modelled with these meta-models and in figure 5.3, the meta-models are shown as ERD's. We can regard meta-model A as the legacy meta-model and meta-model B as the MDE meta-model, or vice versa. As can be seen from the figures, meta-model A structures the elements on a form in groups, whereas in meta-model B, a splitter is used to separate the elements. In the ERD's, we can recognize this by the hierarchical structure of meta-model A, and the more flat structure of meta-model B. Furthermore, the controls on the two forms are also slightly different. Meta-model B uses `labels` and `textboxes`, but meta-model A only uses `fields`, which are a composite of these.

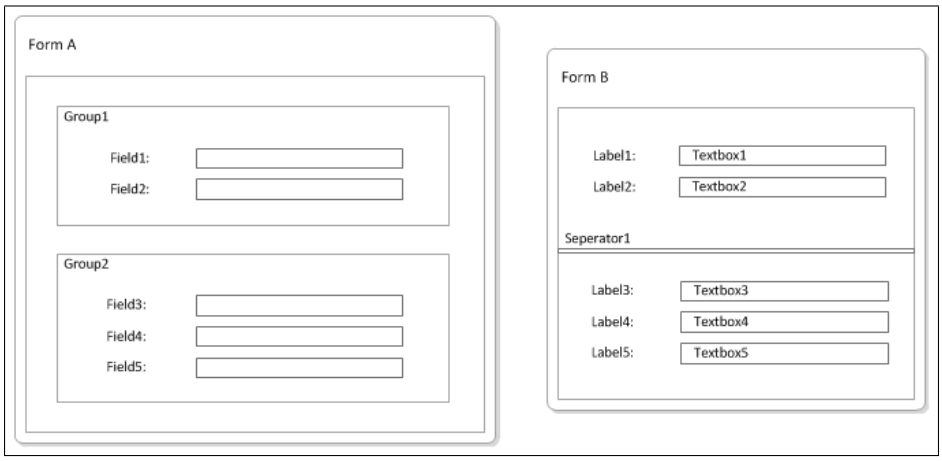

Figure 5.2: Forms example

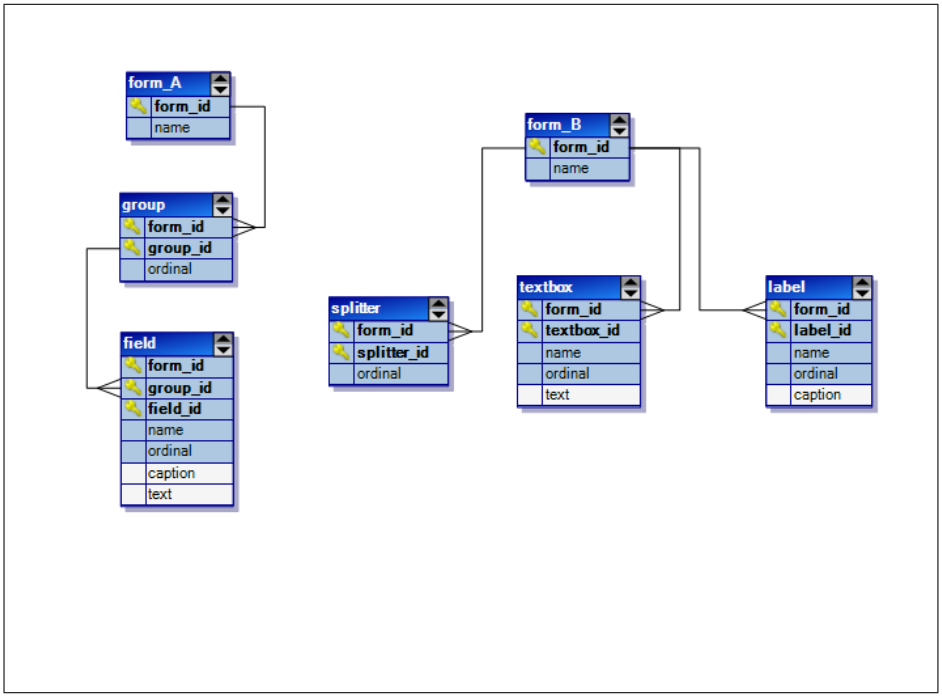

Figure 5.3: Forms example

The first example that we discuss is a transformation from meta-model A to meta-model B. In this situation, meta-model A acts as the legacy meta-model and meta-model B acts as the MDE meta-model. After this, we discuss a second example of the opposite transformation, from meta-model B to meta-model A. As we know from section 3.2, the transformation model is positioned at the level of the meta-models of the source and the target model. This means that the transformation model maps the constructs in the source meta-model to the constructs in the target meta-model. In our case, these "constructs" are the entities, relations and attributes shown in figure 5.3. Recall that by defining the mappings on this (meta) level, the transformation model can be reused for every instance (read: model) of the source meta-model.

5.5.1 From meta-model A to meta-model B

For the transformation of models with meta-model A to meta-model B, we defined the entity and attribute mappings shown in table 5.1. The first table displays the entity mappings, and the second table the attribute mappings. All tuples have a unique id, and the attribute mappings have a reference to the corresponding entity mapping (using the `EM_Id` column). Both tables feature first the target entity or attribute, and in the column thereafter the corresponding source entity, attribute or expression. In this way, the tables can be interpreted as a definition of how to fill the target entities and attributes with the information in the source model.

Entity mappings		
Id	Target	Source
1	form_B	form_A
2	splitter	group_pp
3	label	field
4	textbox	field

Attribute mappings			
EM_Id	Id	Target	Source
1	1	form_id	form_id
1	2	name	name
2	3	splitter_id	id
2	4	ordinal	ordinal * 100
3	5	label_id	field_id
3	6	name	name
3	7	ordinal	group.ordinal * 100 + ordinal * 2
3	8	caption	caption
4	9	textbox_id	field_id
4	10	name	name
4	11	ordinal	group.ordinal * 100 + ordinal * 2 + 1
4	12	text	text

Table 5.1: Example mapping A to B

Let us first consider the entity mappings. The `form_A` and `form_B` entities are mapped directly, as well as the `field` and `label` entities. However, the `field` entity is mapped also to the `textbox` entity. Although the `group` entity from meta-model A and the `splitter` entity from meta-model B have similar semantics – they both separate groups of elements on a form from each other – we cannot map these directly. This is because the cardinality is different here: there is always one more group than there are splitters. Furthermore, in meta-model A, there is always at least one group in every form, but in meta-model B, a form can exists without a splitter. To take care of this, a pre-processor for the `group` entity is used. This pre-processor creates a new entity in the source meta-model: `group_pp`. For each group in a form, except for the first, the pre-processor creates a `group_pp` tuple. In the `group_pp` tuple, the `ordinal` column is copied from the `group` tuple. In this way, the `group_pp` entity and the `splitter` entity, do have the same cardinality. Subsequently, the `group_pp` entity (instead of the `group` entity) is mapped to the `splitter` entity.

From the attribute mappings, we can see that most attributes are mapped directly, but there are some exceptions regarding the ordinal numbers. In order to maintain the same grouping of elements on the form, the ordinal numbers of the elements cannot be copied directly from the source model. This is because, in the source model, the ordinal numbers are always in the context of a `group` entity which is not the case in the target model. In the target model, all the elements on a form (labels, textboxes and splitters) are in the same context: the form itself. This means that, to maintain the same grouping, this needs to be encoded in the ordinal numbers of the elements. For this, we multiply the group

ordinal by 100, and add this as a premium to the ordinals of the elements. This allows for 100 elements per group. Furthermore, since a field is split up into a label and a textbox, the ordinal of a field is multiplied by 2. And since a label usually precedes the textbox, we added zero to the label ordinal and 1 to the textbox ordinal.

The mappings above are clean and simple, because they rely on some important assumptions. The first assumption is that the value for the `form_id` attribute in the target model only has to be determined at the mapping from `form_A` to `form_B`. In the `splitter`, `label` and `textbox` entities, the value for `form_id` should be found by utilizing the relation to `form_B`. Furthermore, the calculation of the ordinal number for the `label` and `textbox` entities employs a lookup to the `group` table in the source model. Recall that these entities are mapped to the `field` entity, and not to the `group` entity. The assumption is that, during execution, the `group` entity can be joined to perform this lookup. The relation between the `group` and the `field` entity should then be used to determine the join conditions.

5.5.2 From meta-model B to meta-model A

For the opposite transformation, of models with meta-model B to meta-model A, we defined the entity and attribute mappings shown in table 5.2. Again, the first table displays the entity mappings, and the second table the attribute mappings.

Entity mappings		
Id	Target	Source
1	form_A	form_B
2	group	form_B
3	group	splitter
4	field	label_textbox_pp

Attribute mappings			
EM_Id	Id	Target	Source
1	1	form_id	form_id
1	2	name	name
2	3	group_id	0
2	4	ordinal	1
2	5	group_id	splitter_id
2	6	ordinal	ordinal
4	7	field_id	id
4	8	name	name
4	9	ordinal	ordinal
4	10	caption	caption
4	11	text	text

Table 5.2: Example mapping B to A

The `form_A` and `form_B` entities are again mapped directly, however, the field entity is now mapped to the result of a pre-processor: `label_textbox_pp`. This pre-processor combines (using some convention) a `label` and a `textbox` in meta-model B, to map this to the `field` entity in meta-model A. The pre-processor also determines after which splitter the label and textbox occurred. Furthermore, the `group` entity is mapped twice. Once to the `form_B` entity, to create a group for every form, and once for each splitter.

The attribute mappings are much simpler this time, because the ordinal numbers can remain the same. All attributes are mapped directly, except in the mapping from `form_B` to group. Here, constant values are used for the id and ordinal number, to ensure this is always the first group on the form. An extra assumption in this transformation is that the `group_id` attribute in the field entity can be determined using the relation to the `group` entity combined with the information in `label_textbox_pp` about after which splitter the elements occurred in the source model.

5.6 Transformation meta-model

In this section we present our transformation meta-model. In figure 5.4, a slightly simplified version of the transformation meta-model is shown. The actual meta-model is more elaborate, mostly to support business rules. In this figure, the essence of the model is more clear.

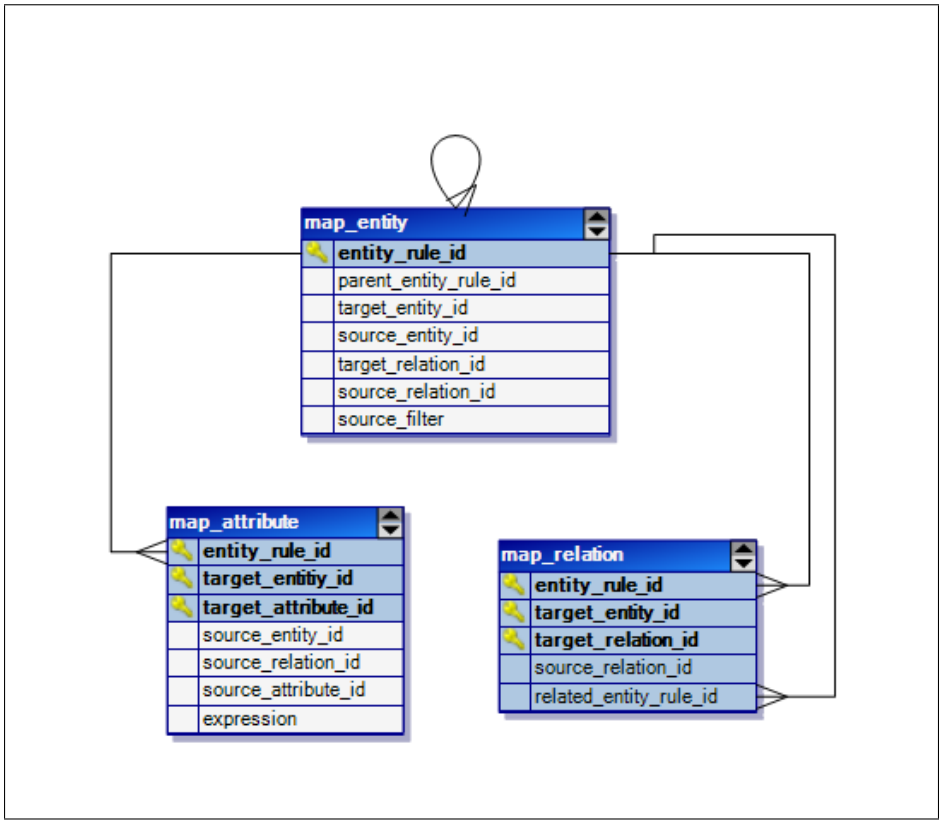

Figure 5.4: Simplified transformation meta-model

The entity relationship diagram in figure 5.4, shows three entities. Blue shaded attributes are mandatory, and the yellow key symbols represent primary key attributes. The top entity: `map_entity` is used to model entity mappings. An entity mapping is a transformation rule, and has a primary key: `entity_rule_id`.

The second attribute is a relation to a parent entity mapping. This is used for mapping weak entities. Mappings of strong entities do not have a parent entity mapping. The next two attributes for target and source entity are only used when this is a mapping of strong entities. The last two attributes do not map entities directly, but via relations. These attributes are only used when mapping weak entities. The `target_relation_id` attribute is used to navigate in the target scheme from a (strong or weak) parent entity to a weak entity. To resemble this step in the source scheme, a similar relation must be chosen in the source model. This is done with the `source_relation_id` attribute. The last attribute of `map_entity`, is `source_filter`. This attribute can be used to apply a filter on the source entity to limit the mapping.

The two entities below `map_entity` both map the attributes of entities. When an attribute (or multiple attributes) represent a relation to a tuple in another entity (foreign key), `map_relation` is used. When this is not case, the attribute is mapped with `map\_attribute`.

The bottom left entity represents the mapping of regular attributes. Obviously, this entity is connected with an entity mapping rule. The two extra primary key attributes, represent a reference to an attribute in the target scheme. The next attribute: `source_entity_id` is copied from the entity mapping rule, and serves as a limiter for the following two attributes. In the attribute mapping, we have three options. The first is to map the target attribute to an attribute in another entity. The second is to map the target attribute to an attribute in the current source entity. And the third option is to provide a custom expression for the target attribute. In the first option (using the `source_relation_id` attribute) we can perform a lookup to another entity in the source model. Here, only relations can be used that have multiplicity 1 (only from foreign key to primary key). Recall that this was needed to determine the ordinal number for the label and textbox entities in the first example.

The last entity, `map_relation`, is to map relations in the target model. Just as with `map_attribute`, `map\_relation` is in the context of an entity mapping. Here, a (foreign key) relation in the target model is mapped to an equivalent relation in the source model. Both relations must have multiplicity 1 (only from foreign key to primary key). To ensure that the referenced entities are also mapped, the corresponding entity rule must be provided using the mandatory `related_entity_rule_id` attribute. Recall that this satisfies our assumptions about the relations in both examples.

5.6.1 Mapping relations

The fact that we opted to explicitly map the relations in the meta-models, has several reasons regarding executability. Although they are not strictly necessary to retrieve all the data in a relational scheme, ignoring relations in the transformation model raises the following issues at execution time:

- A parent entity of a weak entity might not be mapped
- The execution order is not clearly determined
- The attribute mapping (of relation attributes) of a child entity is different from that of its parent entity
- The parent mapping is filtered, but the child entity is not

The first point is not possible with our transformation meta-model, since a weak entity always needs to provide a reference to the parent entity mapping. The second point is also solved with the relations, because the mapped relations form a hierarchy which can be traversed top-down to get the proper ordering for execution. The third point is solved by mapping relations explicitly with `map_relation`. This basically *refers* to another (parent) entity mapping (in which the relation attributes are mapped / translated), instead of providing values for the relation attributes themselves again in the child mapping. The last point is solved with the parent relation in the entity mapping. At execution time, the child source entity is joined with the parent source entity, on which the filter is applied.

5.6.2 Executability and traceability

We have already seen that an entity mapping in the transformation meta-model is also referred to as an entity *rule*. Regarding entity mappings as transformation rules is effective, because an entity mapping is always focussed on a single target entity. Each transformation rule then takes care of one entity in the target meta-model. This means that one can generate the transformation logic in chunks for each target entity. At execution time, it is then a matter of executing the generated code for each transformation rule, where the ordering is determined by the relations hierarchy.

The automatic traceability needs to be provided at execution time, since only then the actual tuples that are subject to the transformation are known. When executing a specific transformation rule, for each tuple that is created in the target model, a trace link can be created as well. The trace link should connect the tuple in the source model with the new tuple in the target model and the current transformation rule (identifier).

For more details on the executability and automatic traceability aspects of the transformation meta-model, we refer to the validation in chapter 7.

5.7 Conclusion

In this chapter we focussed on the model transformation step in our general approach. It answered the second sub-question by exploring examples of schema mappings and providing a transformation meta-model. The requirements for the transformation meta-model were:

- The transformation models should be executable
- The transformation meta-model is tractable
- Automatic traceability can be provided
- Custom analysis can be performed

The first requirement is satisfied by regarding each entity mapping as one transformation rule, and is supported by the mapping of relations. The second requirement is satisfied because the transformation meta-model is limited to one-to-one mappings. This keeps the transformation meta-model comprehensible, while complex transformations can still be performed with the use of custom pre-processors. We reasoned about how the third requirement can be fulfilled in an execution framework. This is validated in chapter 7. Finally, the use of pre-processors, obviously also satisfies the last requirement.

As said before, next to the second sub-question, the chapter also addressed the third sub-question about automatic traceability. This third sub-question is answered fully in chapter 7. We have shown that, in our case, the problem of model transformation can be reduced to that of schema mapping. And to perform schema mapping, we presented our transformation meta-model. Next to entities and attributes, also relations are mapped. This makes it possible to define executable transformation models.

Chapter 6

Thinkwise Software Factory

For the validation of the method (in chapter 7), we use the Thinkwise Software Factory, or TSF, as the MDE tool. Recall that the MDE tool in our method is used in two different ways. On the one hand, the MDE tool is the target of the transformation, and on the other hand, it is used to carry out that transformation. Therefore, before we discuss the validation in chapter 7, it is important to get to know the TSF first. This chapter gives an introduction to the TSF and covers the areas that need to be known for the validation.

Thinkwise Software

Thinkwise Software is a Dutch software company that created and maintains the Thinkwise Software Factory, or TSF. The TSF can be characterized as an MDE tool, in the sense that the primary place of development is a model. The TSF is used in several projects of the clients of Thinkwise (and by Thinkwise itself), and has shown that the development and maintenance costs of information systems can be reduced significantly compared to traditional approaches.

The core of the TSF mainly consists of three parts: the TSF meta-model, the Functionality Weaver and a generic Graphical User Interface (GUI). The TSF meta-model is used to create platform independent application models. These models are stored in a relational database: the TSF repository. The TSF meta-model is discussed in more detail in the following section. Section 6.2 is devoted to the Functionality Weaver, and section 6.3 discusses the generic user interface of the TSF. We conclude the chapter in section 6.4.

6.1 Software Factory meta-model

The TSF uses its own proprietary meta-model to define application models. The TSF meta-model defines the concepts that can be used in defining an application model. This means, in other words, that the meta-model provides the building blocks to create applications with the TSF. In figure 6.1, the components of the

TSF meta-model are shown schematically in the left rectangle. Each component represents a sub-model in the application model.

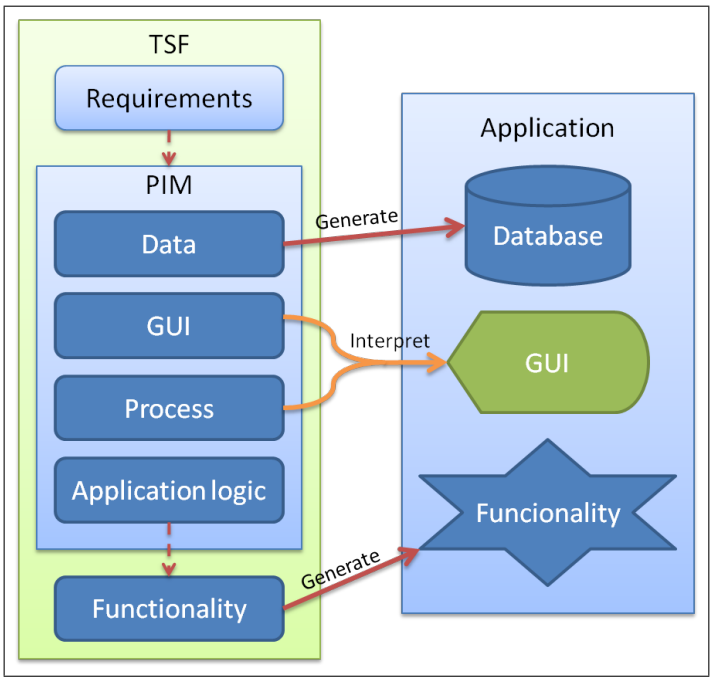

Figure 6.1: Main components of the Thinkwise Software Factory and how they are used

In the figure, we see that the first component is that of the requirements. The requirements model, defines the business, user and system requirements for the (to be developed) application. Once requirements are stated, they can be used in the TSF as a guide to further develop the application model. When new objects are created, while a requirement is selected, a link between the new object and the requirement is created automatically. This can be useful for checking if all requirements are fulfilled, or for traceability.

The next four components in figure 6.1 comprise the Platform Independent Application Model, or PIM [6]. In the data-model, a relational data scheme can be defined in which the client data will be stored. We can see from the figure, that this data-model is used to generate the client database in the application. Next, the GUI model defines how the objects in the model should be visualized in the generic GUI. For example, what type of screen should be used to display the contents of a certain table? Or, what columns should be hidden? Or, what colors should be used in the layout?

The Process model defines how automated process flows should behave across several screens in the application. An example of a process flow is, for instance, that right after a new project is created in a project management system, the GUI automatically opens the project activity screen, to start adding activities. The figure further tells us that this model, along with the GUI model, is not used to generate code. Instead, they are interpreted directly at runtime by the

generic GUI.

Application logic specifications are textual descriptions (therefore still platform independent) of the business functionality. An example would be: "When a new record is being created in a table which contains a date field, fill this with the current date as a default."

The last component in the TSF meta-model is for the functionality. This component is not part of the PIM, because it contains platform specific code templates to implement the business rules. A so called control procedure defines where in the application a specific template should be used. The templates can be parametrized such that they can be reused in different objects. An example would be the code template that sets the current date as a default in all date columns. Even though the columns could have different names (like `begin_date` or `end_date`), the same template can be used to implement this, when the column name is parametrized. The templates are analogous to aspects, and control procedures to the join point specification (pointcut) in aspect oriented programming [24].

6.1.1 Base projects

In the TSF, a model can be denoted as a base project. With a base project, generic functionality can be shared among multiple models (/projects). When a base project is linked to a model, the contents of the base project will be copied automatically into the other model. The TSF supplies several default base projects. There is one, for instance, that generates the DDL statements for the data-model. This base project mainly consists of control procedures and code templates. When the base project is linked to a model, all this is copied to the other model. As a consequence, these control procedure and templates will then also be used in the generation of that model. A base project is also used to implement the framework in the validation in chapter 7.

6.1.2 Meta-modelling

It is interesting to know that the TSF meta-model itself is also stored in the TSF repository. This means that the TSF meta-model is expressed using the TSF meta-model itself. The TSF meta-model therefore functions also as the meta-meta-model. The component of the meta-model that makes this possible, is the data-model. Since a data-model can define any relational scheme, it can also define a scheme which functions as a modelling language. In this way, the TSF meta-model can be used to create other meta-models as well. Since our method requires that a meta-model is constructed for the legacy system, this principle is important. In chapter 7, we see how this is accomplished in the validation.

6.2 Functionality

There are several different ways in which functionality can be incorporated in a TSF application. There are, for instance, the following standard reactive TSF concepts: Default procedures, Layout procedures, and Triggers. But there is also the Task concept, which needs to be called explicitly by the end-user. Default procedures allow to programmatically set default values for specific fields in the GUI. And with Layout procedures, one can control when which fields should be editable or visible. Triggers are analogous to database triggers, and are often transformed to just that. Furthermore, the Default and Layout procedures are automatically called at certain events in the GUI. For instance, when the current record is changed, or when a field loses focus. A Task is a procedure or function that can be called from the GUI by the end-user. A Task can be a stored procedure on the database, but can also be a .NET method.

The Functionality Weaver transforms the data-model and functionality parts of the model into platform specific implementations. Through standard templates and control procedures (acquired via the base project), DDL statements are generated for the data-model. For the functionality, standard interfaces are created for the Default procedures, Layout procedures, and Triggers – which are all automatically created for each table. These procedures have fixed interfaces, accepting all the values of the columns of the current record. This enables the developer to act on these events in a standardized way for all the objects in the model. With the use of client specific code templates and control procedures, business rules are weaved into these procedures.

To summarize: the Functionality Weaver first generates the interfaces for the Thinkwise concepts, and then *weaves* the client specific code into their bodies. In the previous example of the default date to be set to the current date, the default procedure would be used for all tables containing a date column.

6.3 Graphical User Interface

While the data layer and functionality layer are generated from the model, the presentation layer, or GUI, is not. Instead the GUI model (and Process model) are interpreted at runtime. To achieve this, the GUI has both a connection to the TSF repository, and a connection to the (generated) client database. The TSF repository, which contains the model of the application, is used to dynamically built screens to visualize the data stored in the client database. This results in a generic GUI that can be used for all TSF applications, and is available for .NET, Java and Web technology. Figure 6.2 and 6.3 show the .NET and (Java) Web GUI, respectively. There are two major benefits to this approach. One is that the model is no longer just a description of the application, it actually becomes *part* of it. This enforces that the model and generated code is in sync at all times – which would otherwise require a significant amount of discipline of the developers. The second major benefit is that changes in the GUI model or Process model do not require regeneration or deployment efforts. It will immediately take effect when a GUI is restarted. Lastly, this architecture also allows for centralized storage of user preferences and usage logging (at the

meta-level) that is used by any GUI on any computer on any platform. This last point is actually carried out by Thinkwise's Intelligent Application Manager (IAM), which is the production environment for TSF applications and will not be treated in this thesis.

6.4 Conclusion

In this chapter, we provided a brief overview of the basic concepts of the Thinkwise Software Factory. We showed that the TSF meta-model can be broken down into several components, and that base projects can be used for sharing functionality among models. For the data-model and functionality, the Functionality Weaver generates the corresponding code using control procedures and code templates. The GUI and Process model are interpreted directly by the generic GUI at runtime.

Furthermore, due to the data-model component, the TSF meta-model is not only capable of defining models, but also meta-models can be defined. The TSF meta-model can therefore also define itself.

Now that the basic concepts of the TSF are made clear, we continue in the next chapter with the validation of the method using the TSF as the MDE tool. During the validation we, amongst other things, use a base project, extend the TSF meta-model, and use the Functionality Weaver to generate the code for the model transformation.

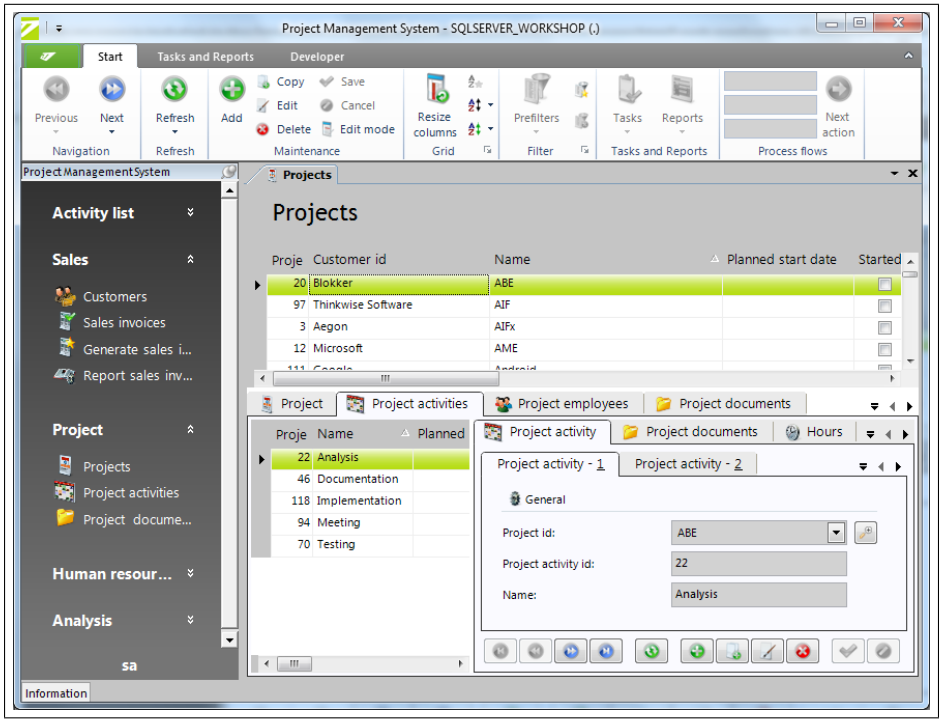

Figure 6.2: Workshop sample project with .NET GUI

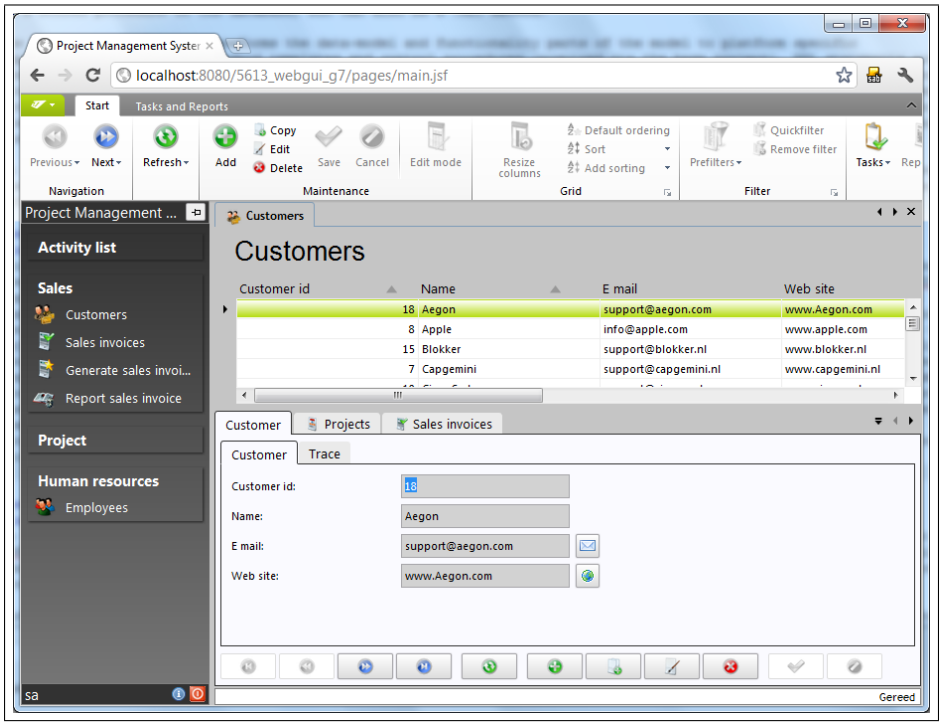

Figure 6.3: Workshop sample project with (Java) Web GUI

Chapter 7

Validation

In the previous chapters we proposed a general method to modernize legacy information systems with MDE. We motivated the design decisions for this method already in theory. However, since a method is only useful if it can be used in practice, a theoretical foundation alone is not sufficient. Therefore, we discuss in this chapter a practical validation of the method.

In chapter 1, we discussed what important aspects of the method need to be validated. To refresh this, we repeated these aspects below:

- Best-effort but complete approach
- Transformation is not restricted to data-model, but also other types of models can be transformed (e.g. GUI-model, Process-model, etc.)
- Automatic traceability
- Easy to use for someone with experience with the TSF
- Allows for the use of custom code
- Extensible for new legacy systems

The validation is carried out with the Thinkwise Software Factory (TSF) as the MDE tool, which is introduced in the previous chapter. In the validation, we perform a pilot project to modernize the Microsoft Access sample application: Northwind. Since the TSF is our MDE tool, this means transforming the model of the Northwind application such that it fits in the TSF. Our method prescribes that, to accomplish this, we need to first construct a meta-model for Access. For this meta-model a new new application is created (with the TSF), called the Access Upcycler. The Access Upcycler is responsible for extracting and storing models from Microsoft Access, as well as transforming them to TSF models. In figure 7.1, the Access Upcycler is shown in relation to the TSF.

In the figure, the TSF occurs twice: once as the Master SF, and once as the Client SF. In the Master SF, there are three models: SF, COBOL Upcycler, and Access Upcycler. The SF model, models the TSF itself – this makes it the meta-meta-model. The other two models, define Upcycler applications for COBOL and Access respectively. (The COBOL Upcycler is depicted only to illustrate

that there can be multiple Upcyclers; it was not developed in the validation.) The Upcycler models use the base project: Upcyclers. This base project is used to share generic modernization functionality between Upcyclers.

A red arrow in figure 7.1 depicts an instantiation of a model in the TSF. In this way, the Master and Client SF are both instantiations of the SF model. (We use two instantiations to separate the meta-modelling in the Master SF from the "normal" modelling in the Client SF.) To illustrate the normal use of the Client SF, a model of a Human Resource Management application is shown as an example. The HRM application that is generated from this HRM model, is shown in the top right corner of the figure. However, our purpose is to use the Access Upcycler to create a model in the Client SF. Therefore, the Access Upcycler is also instantiated from its model in the Master SF. The purpose of the Access Upcycler is to load a model from Microsoft Access, transform this, and put it in the Client SF. This last property is symbolized by the arrow towards the Client SF. A sample Access application, the Northwind sample from Microsoft, is shown in the bottom left corner of the figure. The Northwind application is loaded in the Access Upcycler, and transformed into the Northwind model in the Client SF. From this model a new Northwind application can be instantiated, and further development can be carried out with the Client SF.

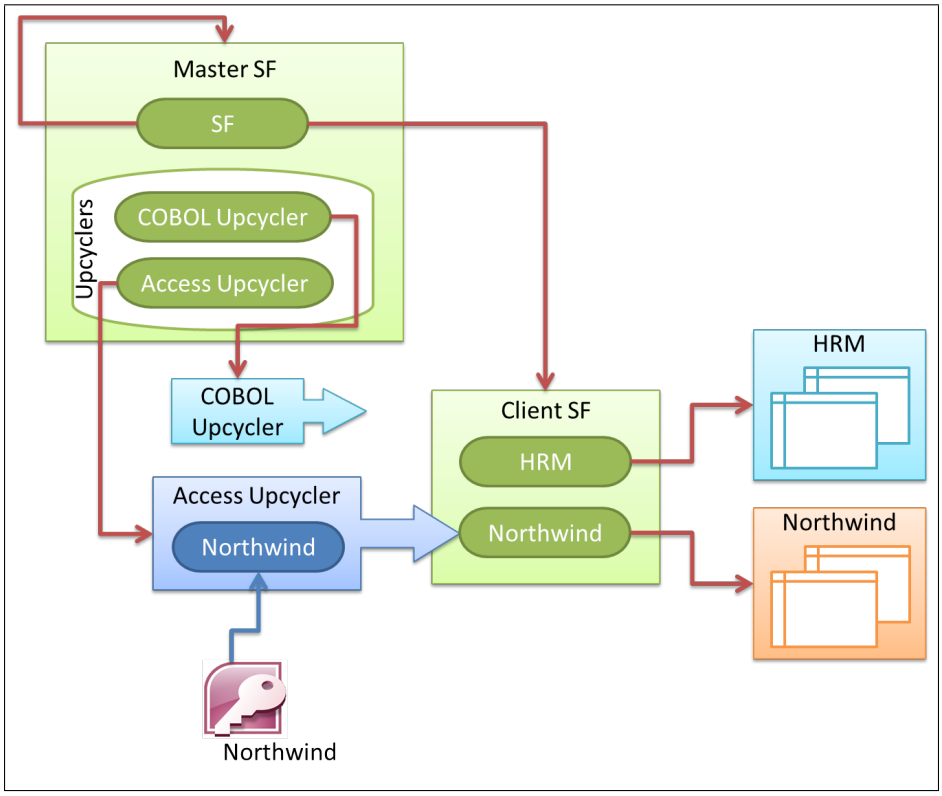

Figure 7.1: Master and Client SF

Part	Component	Applicable to
Framework	Upcyclers base project	All Upcyclers
	Abstract pre-processor	All Upcyclers
	Transformation meta-model	All TSF projects
Access Upcycler	Access meta-model	All Access (2010) applications
	Access extraction module	All Access (2010) applications
	Transformation model	All Access (2010) models
Northwind → TSF (Pilot project)	Northwind model in Access Upcycler	Only Northwind
	Transformation to TSF	Only Northwind
	Trace links	Only Northwind

Table 7.1: Different parts in validation

Parts

In order to validate the method, several different activities need to be done that each have a different level of generality. To clarify what is built / carried out, and for what purpose, we divided the activities in three parts which are shown in table 7.1. The table shows that the framework is the most general part, since its components are applicable to all Upcyclers. The abstract pre-processor contains general read, write and traceability functionality used by pre-processor implementations. Furthermore, since model transformations are not only useful in Upcyclers, the transformation meta-model is applicable to all TSF projects. The framework is used to support the creation of Upcyclers. The Access Upcycler is an example of such an Upcycler, and is specifically designed to upcycle Access (2010) applications. As a consequence, the applicability of the Access Upcycler is narrower than that of the framework, and consists of all Access 2010 applications. Finally, the Access Upcycler is used in a pilot project to upcycle Microsoft's Northwind sample application into the TSF. The model, transformation, and trace links in this part are only relevant for this specific project.

To summarize, we have (1) the framework that contains the most generic components that are applicable to all Upcyclers, (2) the specific Upcycler for Access that is applicable to all Access (2010) applications, and (3) the pilot project that only applies to the Northwind application.

Overview

In the following sections, we discuss the three parts of the validation in more detail. The development work is discussed bottom-up from concrete to abstract. We begin with an introduction of the pilot project, which serves as a concrete starting point. We advance then into the creation of the Access Upcycler. This is more general, but is limited to Access applications. In the section thereafter, we discuss how the framework is constructed that supported the creation of the Access Upcycler. This is the most general part, and after this, we look back again at our concrete pilot project to discuss the results.

7.1 Pilot project

The pilot project in our validation is the Microsoft sample application: Northwind. It is frequently used to demonstrate the functions of Access. Therefore, it is a small but representative instance of an Access application. The Northwind application is an application that processes information about the fictive company: Norhtwind Traders. It is a company that imports and exports food product around the globe. In figure 7.2 a screen-shot is shown of the list of suppliers.

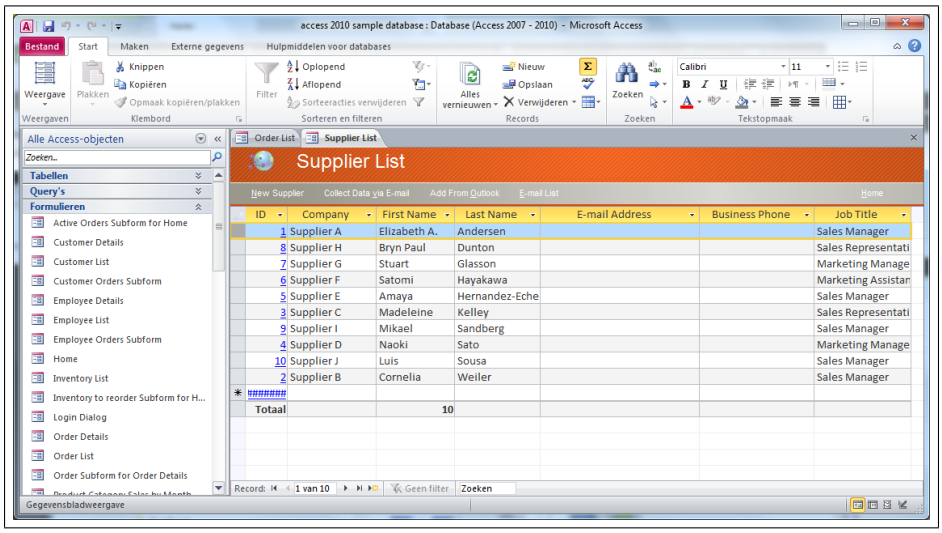

Figure 7.2: Northwind sample application

Our goal is to transform as much as possible from this application into a TSF model. To accomplish this, we create an Access Upcycler that can be used for all Access applications. This is discussed in the next section, but first we need to investigate what objects there are that need to be transformed.

In Access, the following types of objects can be used:

- Tables
- Queries
- Forms
- Reports
- Macro's
- Modules

The Table objects in Access represent the (relational) data model of the application, and support columns, primary keys and foreign keys. The Queries are analogous to views in relational databases, and are based on a SQL query. Furthermore, the columns of a view are also explicitly defined in Access, in the same way as a Table object. The Form objects in Access define the GUI model,

and are bound to a Table or Query object. A Form, furthermore, contains user controls like Labels, TextBoxes and ComboBoxes to enable the end-user to read and write data. The same controls we can find on Report objects, but here they are obviously used for reading data only. Macro's in Access are behavioural constructs which can be used to program simple actions, like opening and closing Form objects. Modules contain Visual Basic for Applications (VBA) code, in which the rest of business logic is implemented. Forms (and controls on Forms) expose events on which event-handlers can be created in the form of a VBA Sub routine.

In this section we explored which concepts are used in Access applications. Recall that our goal in this validation is to transform the Northwind (Access) application into a TSF model. Since the concepts of Access application are clear, we can now focus on how to transform these applications into TSF models. This transformation is to be carried out with a so-called Access Upcycler. How this Access Upcycler is created is discussed in the next section.

7.2 Access Upcycler

In this section, we discuss the implementation of the Access Upcycler. Recall that all Upcyclers are created with the Master SF, so this is also the case for the Access Upcycler. The following subsections describe the different parts of the implementation of the Access Upcycler. First, we discuss the meta-model of Access applications followed by the extraction mechanism used to extract the application model from Access.

7.2.1 Access meta-model

As we know from chapter 4, our approach to modelling legacy systems, is to create a specific meta-model for each type of legacy system. The type of our legacy system is in this case Access, so we need to create a meta-model for Access applications. This meta-model should contain the concepts of Access discussed in the previous section. Finally, the meta-model is created in the Master SF, which means that the meta-model is represented by a relational schema.

Because Access provides some good programming interfaces, we were able to efficiently derive the meta-model for it. For the data-model, we created a so called OleDbConnection to the Access database. This connection can be used to query the data inside the access application using regular SQL. It, however, also features the GetOleDbSchemaTable method. With this method, the entire meta-model of the database-part of Access could be extracted. This took care of the Table and Query objects.

The other Access concepts are represented by an object-oriented structure in the Access application. The meta-model for these concepts was derived by employing the Microsoft Office Interoperability (Interop) [25] functionality. This Interop functionality allows one to programmatically access the objects in a given Access application. This provides a convenient way to retrieve the objects, and with the use of reflection, the classes and properties of these objects can

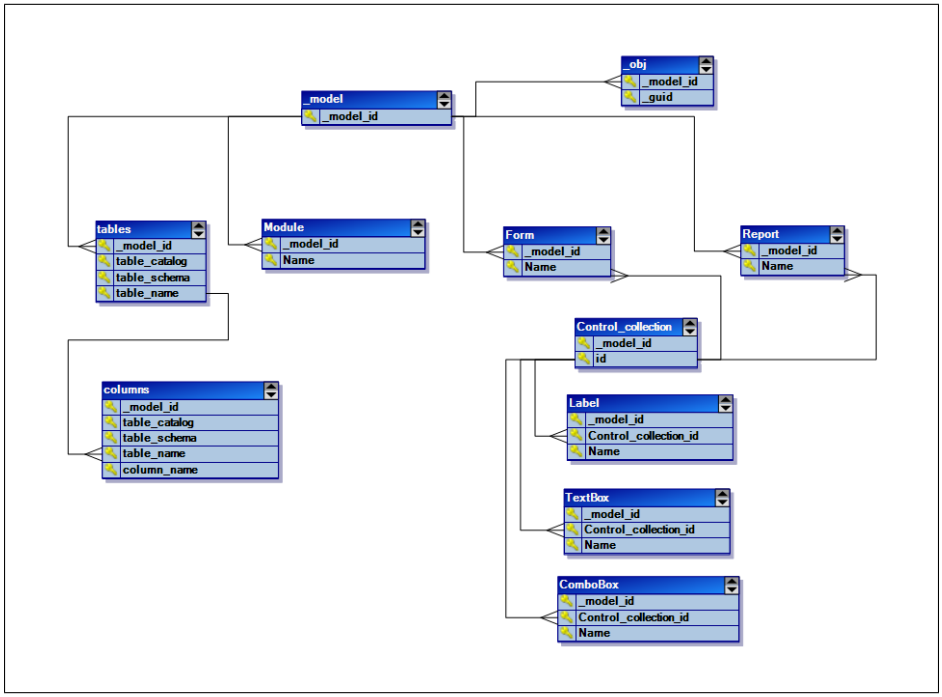

Figure 7.3: Sample of extracted Access meta-model

be extracted. In this way, the meta-model for the Access concepts: Forms, Reports, Macro's and Modules was extracted programmatically.

In figure 7.3, a sample of the created meta-model is shown. The `_model` and `_obj` entities come from the framework, such that all Upcyclers have these entities. The creation of these entities is discussed in section 7.3. The `_model` entity is to provide a context for instances of the meta-model. This makes it possible to use a single Access Upcycler for multiple Access applications. The `_obj` entity is to provide a single list of all the objects in the model, and is also used for the trace links. For simplicity, only the primary keys are shown of the entities.

One thing to note is the way the user controls are modelled. In the object oriented structure of Access, the `Form` and `Report` object both have a `Controls` property. This property is a collection of the user controls on the `Form` or `Report`. Furthermore, the same type of controls that can be used in a `Form`, can also be used in a `Report`. To represent this in a relational meta-model, we introduced the entity: `Control_collection`. In this way, a specific control, for instance a `TextBox`, is always part of a control collection, which in turn is owned by either a `Form` or a `Report`.

Using the constructed meta-model, the Master SF was used to generate the database for the Access Upcycler. This database is now ready to store models of Access applications. The extraction of Access models is treated in the next subsection. In figure 7.4, the TSF GUI is shown on the (still empty) Access Upcycler.

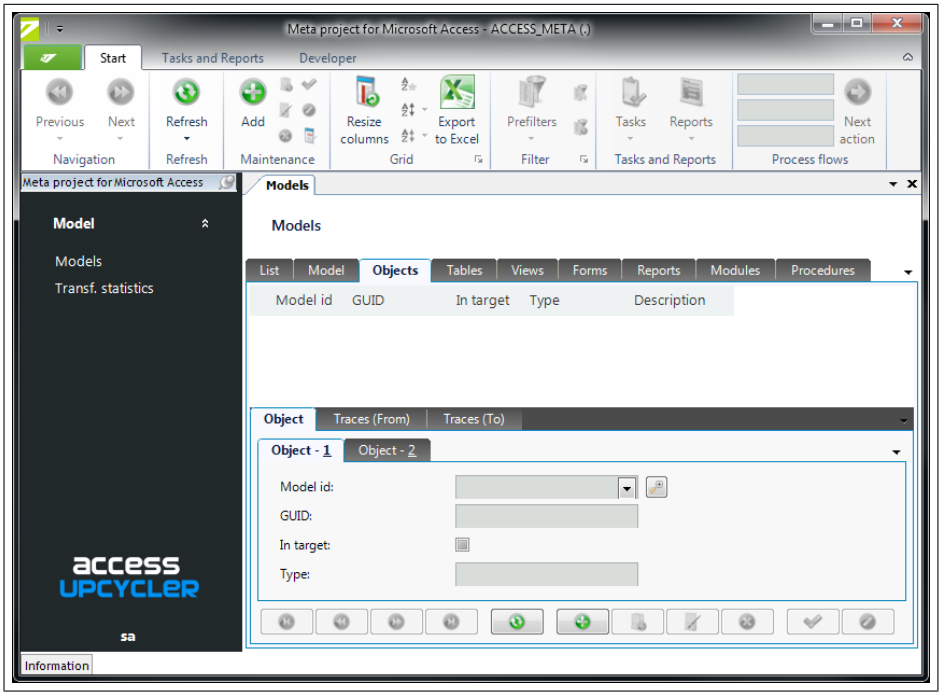

Figure 7.4: Empty Access Upcycler visualized by TSF GUI

7.2.2 Access model extraction

To extract the model of an Access application we created a .NET task in the Access Upcycler that accepts an Access file (.accdb) as a parameter. Within the task, the same techniques that were used in the extraction of the meta-model, are also used for the extraction of the model. An OleDbConnection to the (internal) database of the Access application is used to extract the data-model of the Access application. The other objects in the Access application, such as Forms and Modules, are extracted using the Microsoft Office Interop functionality. Since the meta-model of the Access Upcycler, at this point, is already in place and is tailored specifically for Access models, the extraction of the model is relatively straightforward.

After the extraction task is used for a specific Access application, the Access Upcycler contains a model of that application. Recall that the meta-model of this model is that of Access applications, created in the previous subsection. The model is therefore not yet suitable for the TSF, and needs to be transformed. However, as was discussed in chapter 5, only transformations between entities with the same cardinality can be performed with the transformation model. If the cardinalities do not match, a preprocessor is needed to accomplish this. In the next subsection we discuss an example of a preprocessor, and after this, the transformation model for Access models is shown in subsection 7.2.4.

7.2.3 Routines pre-processor

The extraction task in the Access Upcycler, creates a single record in the Module table for each Module object in the Access application. In this record, the Visual Basic source code is stored in a single field. However, a single Module can contain multiple Functions or Sub routines. Our goal is to create Application logic specifications (a TSF concept) for the business logic contained in these Functions and Subs. But since we can only transform entities with the same cardinality, we cannot use the module entity directly for this. Instead, we need to extract the Functions and Sub-routines contained in them first, and then use the result in the transformation. This is accomplished with what we call a pre-processor. It pre-processes the source model (the Modules entity) such that the result can be used in the transformation. In a complete Access Upcycler, we probably need many preprocessors to successfully transform all the concepts. However, for the validation of our method, we implemented only the routines preprocessor. Because this is a representative example of a necessary preprocessor, it is sufficient for our purposes.

The routines pre-processor is implemented as a custom .NET task in the Access Upcycler. It uses a generic component from the framework to process every record in the modules table one at the time (for a specific model). It analyses the source code field for each record, by searching for function and sub definitions. For every found routine, it invokes an Emit method in the framework module. This, in turn, produces a record in a manually created Routine table, and automatically creates a trace link from the input tuple (in the Module table) to the output tuple (in the Routine table). This functionality makes the traceability transitive over preprocessors that use this framework module.

7.2.4 Transformation model

Since both the Access and the TSF meta-model reside in the Master SF, the transformation model between these meta-models is also defined in the Master SF (see also chapter 5). For the validation, we limited the transformation model to only transform the data-model and the VBA routines. It should be clear that for a complete Access Upcycler also the other concepts like Forms, Macros, Reports, etc., should be transformed. The data-model and the VBA routines are just representative examples that are sufficient for our validation. In the remainder of this section, we discuss some details in the mappings for the data-model and VBA routines.

The data model (the Tables, Queries and keys) of the Access meta-model is relatively straightforward to transform, since the Access and TSF data models are both relational. In figure 7.5, the mapping for the Tables is shown. In the top list, the mappings on the strong entities in the models are shown. The opened tabpage: Column mappings, lists all the columns of the target entity (in this case: `tab`). For each column in the target entity, a value can be selected from the source entity or a lookup table, or – as is shown in the figure – a custom expression can be used. From the figure we can see that the `tab_id` field in the tab entity is filled with the expression: `replace(t.table_name, ' ', '_')`. In this expression, `t` is the alias for the source entity. This means the `tab_id`

column is mapped to the `table_name` column in the `tables` entity, but a replace function is used to remove spaces in the name. This was necessary because the TSF does not allow the use spaces in id's. The list further shows that the `type_of_table` is derived from the `table_type` column using a case statement. Both columns express the fact if a the current table is a view or a regular table, but both use a different encoding to do this (integers vs. strings).

In figure 7.6, the mapping between the application logic specification table `appl_logic` and the `routines` table is shown. Here, a where clause is used to limit the tuples in the source system. This is done to only use the Sub routines (type = 0) that are event handlers for the AfterUpdate event. We can see from the column mappings that the application logic description is filled with the source code of the routine. This code will be manually translated later by a developer. Notice further, that the `appl_logic_id` is mapped to a concatenation of the name of the module, the name of the routine, and the parameter types of the routine.

In the TSF, an application logic specification can be linked to model objects. The event handlers in the pilot project are event handlers of (controls on) a Form in Access. Since a Form, in turn, can be connected to a table (via its record source), we can also map this relation. This is shown in figure 7.7, and 7.8. The red and blue dashed lines in figure 7.8 represent the mappings. Notice that only the entities `appl_logic` and `routine` and the entities `tab` and `tables` are mapped directly. These are the mappings of the strong entities, and therefore have a different color each. The other mappings are mappings of weak entities, and are all mapped via the references. These mappings have the same color as their parent mapping: red. This information is used by the framework to generate the appropriate joins between the tables when transforming an Access model. Furthermore, this also means that the mapping of the `appl_logic_id` column does not need to be specified again for these entities. Instead, the concatenation as specified in in the previous (parent) mapping will be copied automatically. Lastly, the figure illustrates that for the foreign key mapping between `static_assignment` and `tab` (the third red mapping), the blue mapping is needed. This dependency is also shown in 7.7, where the mandatory field `Fk create rule` points to the `tab / tables` mapping: 19.

The examples above should clarify what the transformation model looks like, and how it is created. We described some details of the mappings of the data-model and VBA routines. However, it should be clear that these descriptions do not entail *all* the details of the transformation model that we created for the validation. Since this would be too excessive, we opted to only discuss a few details to illustrate the transformation model.

With the Access meta-model, the Access extractor, a preprocessor and the transformation model in place, the Access Upcycler is defined sufficiently for the validation. The framework and the Master SF is further used to generate the source code for the transformation and traceability. The framework also provides a generic task to export a model in an Upcycler to a Client SF. This task uses the generated transformation code to accomplish this.

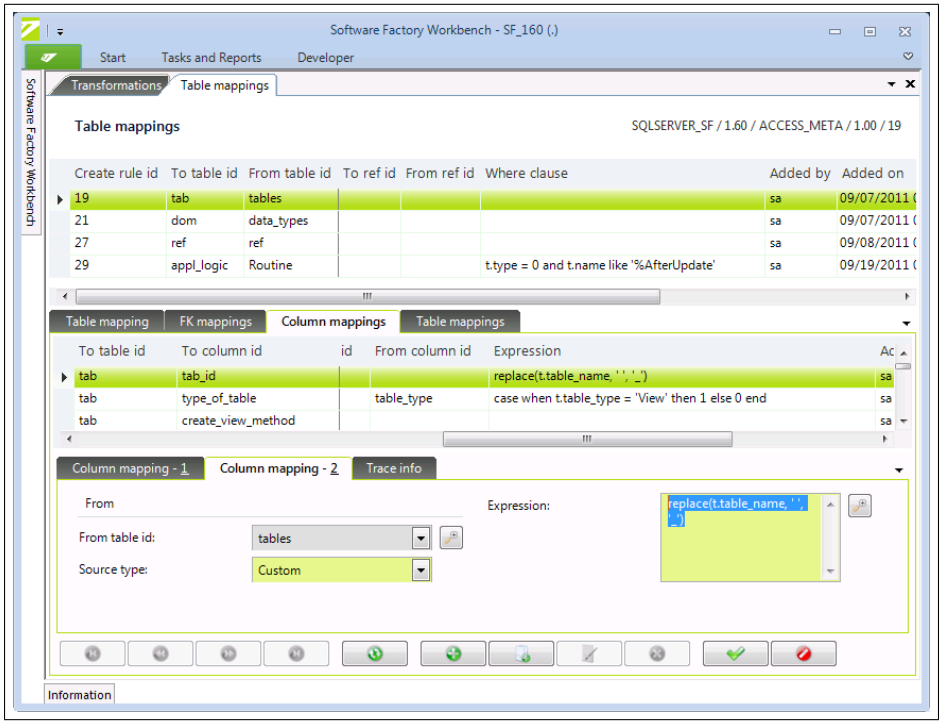

Figure 7.5: The mapping between tab (TSF) and tables (Access)

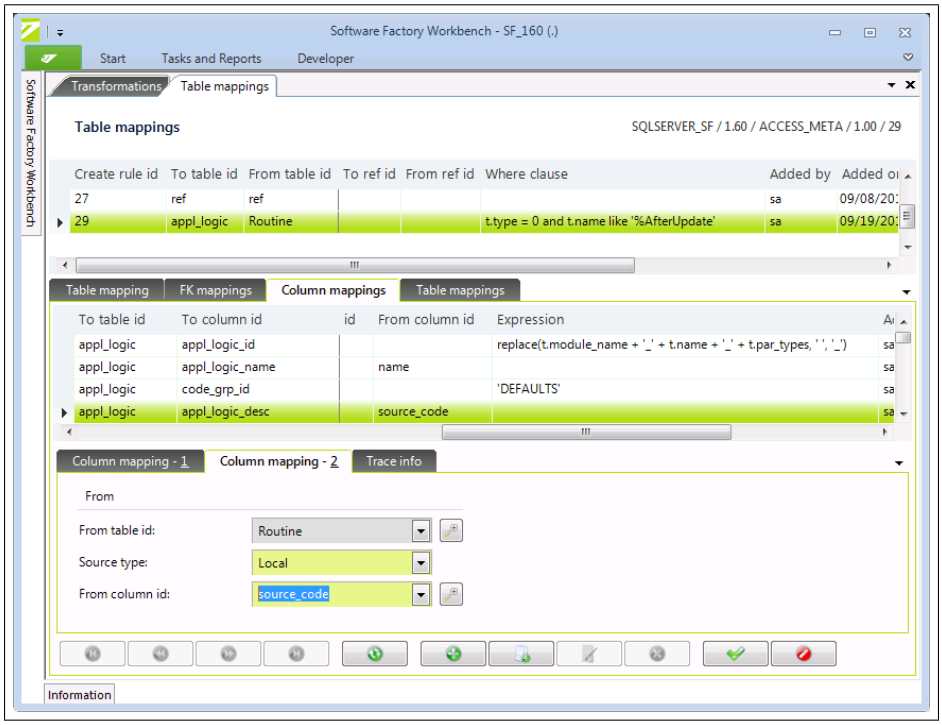

Figure 7.6: The mapping between application logic specifications (TSF) and routines (Access)

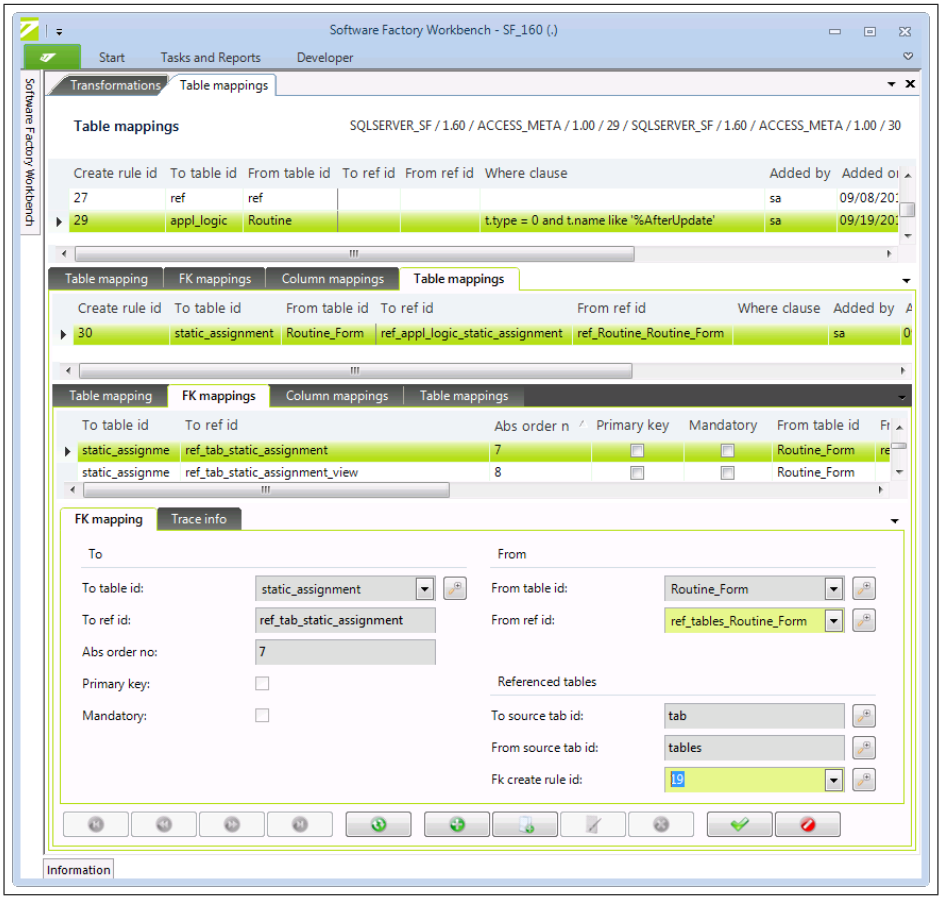

Figure 7.7: The mapping between static assignments (TSF) and forms (Access)

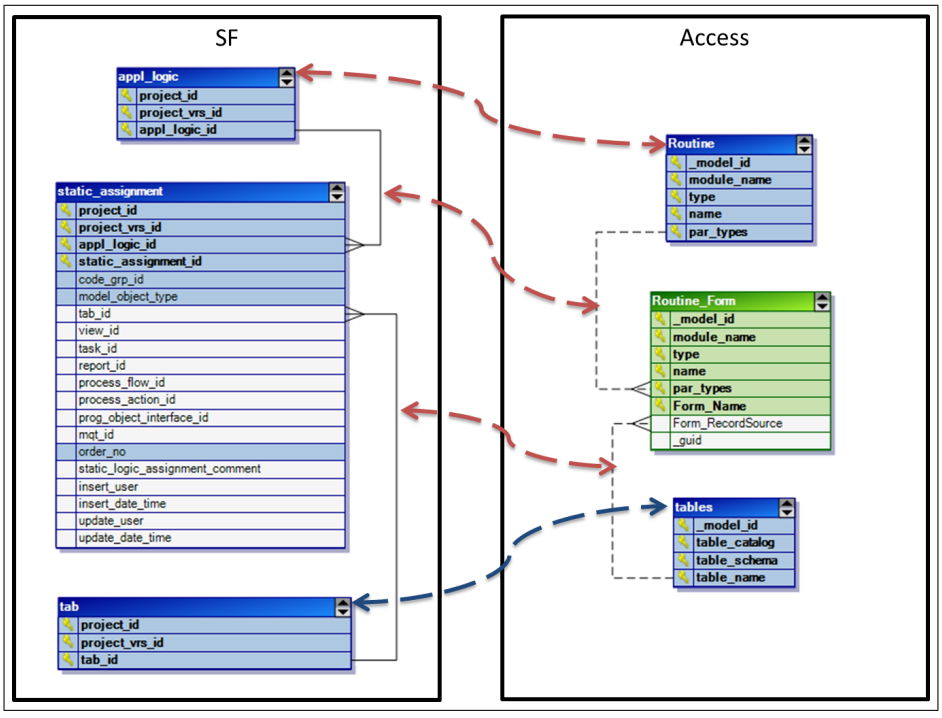

Figure 7.8: The mapping between application logic, static assignments, tabs (TSF) and routines, forms, and tables (Access)

7.3 Framework

In the previous section, we have already seen and used the features of the framework in developing the Access Upcycler. In this chapter we discuss how these features are implemented in the framework.

7.3.1 General entities

Using a base project in the Master SF, the entities `_model`, `_obj` and `_trace` are automatically added to an Upcycler. The entities are prefixed with an underscore to denote that they are part of the framework. The creation is accomplished using a so-called dynamic model procedure. With such a procedure it is possible to programmatically modify the model when the project is generated in the Master SF. Since the TSF uses a relational database to store its models, this procedure consists of SQL statements that create the above mentioned entities in the data model of the Upcycler. It furthermore creates references from every other entity in the meta-model to the `_model` entity. This creates a context for all the entities, such that multiple models can be stored in the Upcycler.

The `_obj` entity is used to create a single list of all the objects in a model. On every entity that the developer creates, an insert trigger is automatically generated that creates a record in the `_obj` table. In this way, when an arbitrary object is created during the extraction of the model from the legacy system, it is ensured that a `_obj` record is created as well. In our pilot project, for example, when a `module` record is produced by the Access extractor, an `_obj` record is created automatically with the same GUID (Globally Unique Identifier). This is useful to get an overview of all the objects in the model, and is also used for the trace links. The trace links are stored in the `_trace` table which has two references to the the `_obj` entity, one for the origin and one for the destination object. The `_trace` entity furthermore has a field to denote which transformation rule was responsible for the trace link.

7.3.2 Transformation generation

Arguably the most important part of the framework is the generation of the transformation code from a transformation model. The transformation meta-model, as discussed in chapter 5, is created in such a way that instances of the meta-model are executable. This means that we can deterministically derive the transformation code from the model. This is accomplished using a so-called dynamic control procedure in combination with (SQL) code templates. The control procedure specifies for each transformation rule (entity mapping), which templates with which parameters are used.

The control procedure in principle generates an SQL select query for each entity mapping. In the from clause of the query, the source entity is used. The select clause consists of a column for each column in the target entity. The value of the columns are derived from the column mappings and the foreign key mappings. In the case of a foreign key mapping, a join is performed on a sub-query which

performs the entity mapping of the referenced entities. The join columns are derived from the foreign key columns between the current and the referenced entities. In the case of a regular column mapping, the value can be a column in the source entity (local) a column in a lookup table, or a custom expression. An example of the result of such a generation for the link between a VBA routine and a table is shown in listing 7.1.

Listing 7.1: Generated transformation procedure for static assignments.

```
select
  t._guid                                as _from_guid
 ,cast(ref_Routine_Routine_Form_29.project_id as varchar(100))
                                         as project_id
 ,cast(ref_Routine_Routine_Form_29.project_vrs_id as varchar(100))
                                         as project_vrs_id
 ,cast(ref_Routine_Routine_Form_29.appl_logic_id as varchar(100))
                                         as appl_logic_id
 ,cast('DEFAULTS' as varchar(100))       as code_grp_id
 ,cast(0 as tinyint)                     as model_object_type
 ,cast(ref_tables_Routine_Form_19.tab_id as varchar(100))
                                         as tab_id
 ,cast(null as varchar(100))             as view_id
 ,cast(null as varchar(100))             as task_id
 ,cast(null as varchar(100))             as report_id
 ,cast(null as varchar(50))              as process_flow_id
 ,cast(null as varchar(100))             as process_action_id
 ,cast(null as varchar(100))             as prog_object_interface_id
 ,cast(null as varchar(100))             as mqt_id
 ,cast(10 as int)                        as order_no
 ,cast(null as varchar(3000))            as static_logic_assignment_comment
 ,cast(null as varchar(100))             as insert_user
 ,cast(null as datetime)                 as insert_date_time
 ,cast(null as varchar(100))             as update_user
 ,cast(null as datetime)                 as update_date_time
from Routine_Form t
 join
 (
   select
      t._model_id      as join__model_id
     ,t.module_name    as join_module_name
     ,t.type           as join_type
     ,t.name           as join_name
     ,t.par_types      as join_par_types
     ,null             as project_id
     ,null             as project_vrs_id
     ,replace(t.module_name + '_' + t.name + '_' + t.par_types, ' ', '_')
                       as appl_logic_id
   from Routine t
   where t.type = 0 and t.name like '%AfterUpdate'
 ) ref_Routine_Routine_Form_29
   on  ref_Routine_Routine_Form_29.join__model_id   = t._model_id
   and ref_Routine_Routine_Form_29.join_module_name = t.module_name
   and ref_Routine_Routine_Form_29.join_type        = t.type
   and ref_Routine_Routine_Form_29.join_name        = t.name
   and ref_Routine_Routine_Form_29.join_par_types   = t.par_types

 join
 (
   select
      t._model_id    as join__model_id
     ,t.table_name   as join_table_name
     ,null           as project_id
     ,null           as project_vrs_id
     ,replace(t.table_name, ' ', '_') as tab_id
   from tables t
 ) ref_tables_Routine_Form_19
   on  ref_tables_Routine_Form_19.join__model_id  = t._model_id
   and ref_tables_Routine_Form_19.join_table_name = t.Form_RecordSource

where t._model_id = @_model_id
```

As we can see in listing 7.1, the transformation logic is represented by a generated SQL select query. In this case, it represents the transformation from the `Routines_Form` entity in Access to the `static_assignment` entity in the TSF. This transformation was also depicted in figure 7.8. There are several interesting things to observe in this query. The first thing to note is that the from clause begins with the source entity of the mapping: `Routines_Form`. This means this query is meant to be executed on the source schema. Furthermore, the columns in the select clause of the query (apart from the first column) are precisely the columns of the `static_assignment` entity. This means the result of the query will "fit" in the target entity in the TSF. Also, all the columns are cast to the datatypes of the corresponding columns in the target entity to prevent datatype issues during execution.

Another observation is that the from clause contains two joins with sub-queries, which are used for the values in the main select clause. Both sub-queries consult another table in the source scheme, namely `Routine` and `tables` (see again figure 7.8). Since these tables, in turn, are also mapped to their corresponding tables in the target schema (`appl_logic` and `tab` respectively), the sub-queries perform these mappings. We can see on lines 37-38, for example, that the `appl_logic_id` is defined by a concatenation of the module name, the routine name and the parameter types of the VBA routine. The join columns for the sub-queries are defined by the corresponding foreign keys in the main source table of this query: `Routines_Form`. This shows us, for example, that to join the `tables` entity, the `Form_RecordSource` column of `Routines_Form` is matched to the `table_name` column of `tables` on line 59. Finally, we can see that the concatenation for the `appl_logic_id` column is not performed again in the main select clause (where the `appl_logic_id` is also present). Instead, the result of the sub-query is used on line 7.

7.4 Pilot results

Now that we have discussed the creation of the Access Upcycler and the framework, it is time to put it to work. Recall that our pilot project was to upcycle the Northwind Access application. In this section, we discuss the results of that process.

7.4.1 Model extraction

The extraction of the model of the Northwind application was accomplished with the extraction task that we already discussed in subsection 7.2.2. The task accepted the Northwind Access file path as a parameter, and subsequently extracted the Northwind model. In figure 7.9, a screenshot is shown from some of the Forms that were extracted from the Northwind application. In the figure, the Textboxes tabpage shows the Textboxes used in the currently selected Form.

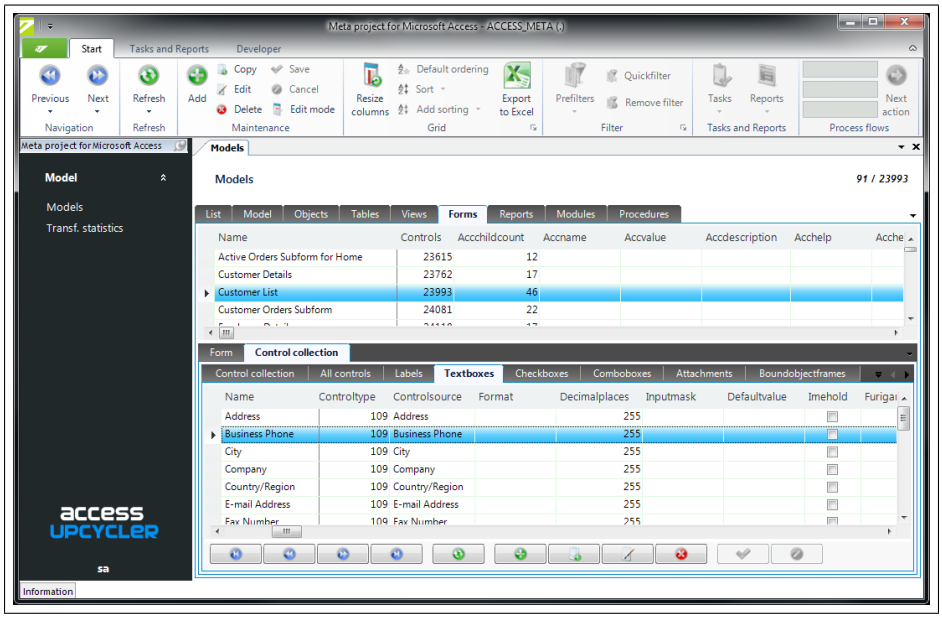

Figure 7.9: Extracted Forms from Northwind

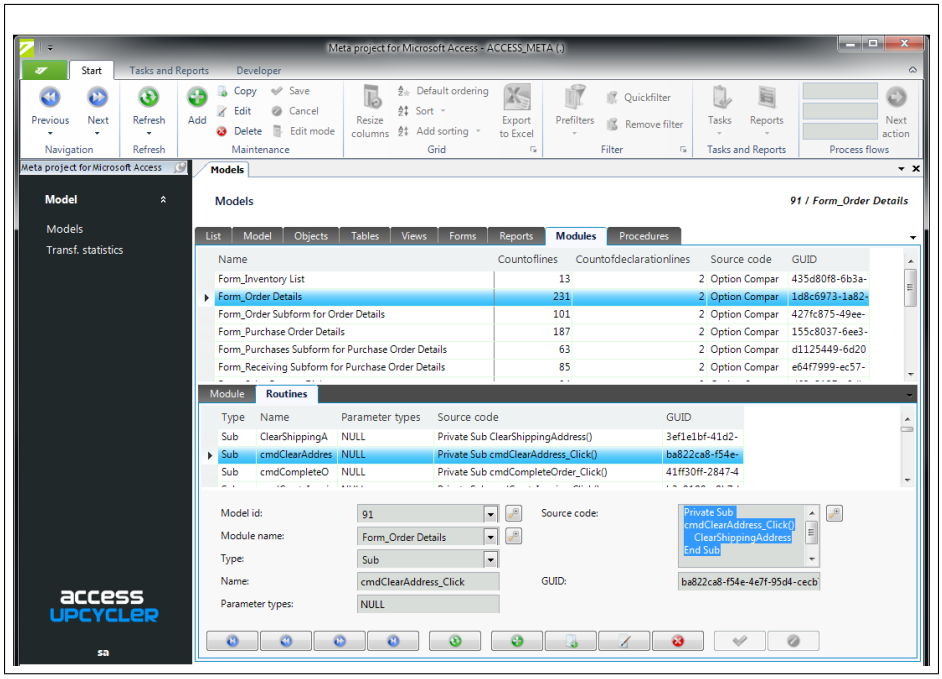

Figure 7.10: Extracted Modules and derived Routines from Northwind

7.4.2 Routines pre-processor

The Northwind application uses VBA Modules to define business functionality. Since these Modules cannot be mapped directly to Application logic specifications in the TSF (see section 7.2.3), we used the Routines pre-processor to split the Modules into individual Function and Sub Routines. The pre-processor was executed manually, but this can also be done automatically before when a transformation is started. In figure 7.10, the derived Routines are shown. The Routine: `cmdClearAddress_Click` is selected, which is – based on the naming convention in Access – an eventhandler for the click-event on the CommandButton: `ClearAddress`. In the Source code field, we can see that a call is made to the `ClearShippingAdress`, which is another Routine.

7.4.3 Transformation

Executing the transformation was done by running the framework task: Export to TSF. With this task, a connection is created with the Client SF. It then calls the transformation procedures that were generated by the framework earlier. The order in which the procedures are called is determined by the dependencies between the target entities. The table entities, for example, are created before the column entities, since a column always belongs to a table. This is necessary to prevent referential constraint errors in the Client SF.

The tuples that the transformation procedures produce, are inserted in the corresponding target tables in the Client SF, using the just established connection. A portion of the transformed data model in the TSF is shown in figure 7.11. Furthermore, a screen-shot of the created application logic specifications is shown in figure 7.12. Finally, in figure 7.13, the proxy objects in the Client SF for the untransformed objects are shown.

End product

For the sake of completeness, we generated the Northwind project in the TSF, and manually copied over (a part of) the data from the original Northwind application to the new database. This allowed us to visualize our newly created application with the Thinkwise GUI. The result is shown in figure 7.14. In the figure, the same data is shown as in the figure 7.2, which was our introduction to the original Northwind application.

7.4.4 Traceability

The trace links generated by the framework during the transformation can be shown in a specially designed Business Intelligence Cube. This cube is shown in figure 7.15. On the vertical dimension in the cube, the object types in Access are shown. On the horizontal dimension, the object types of the TSF are shown into which the Access objects are transformed. The first column in the cube, without a name, denotes the amount of objects that were not transformed. The

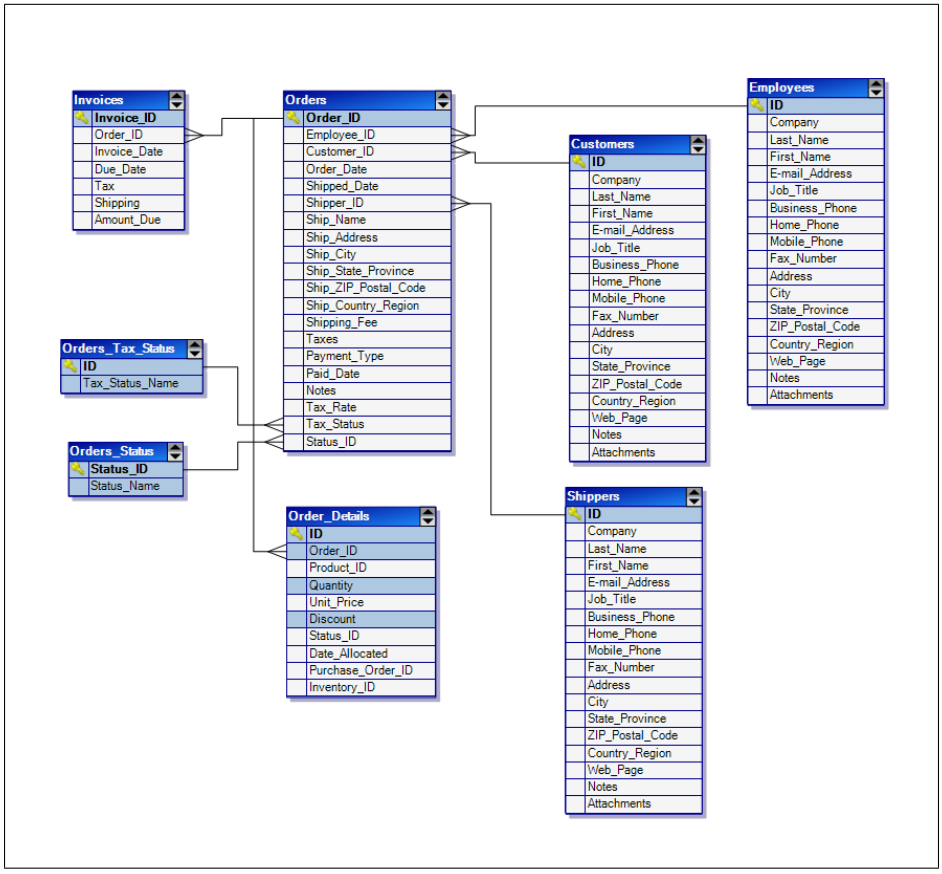

Figure 7.11: Transformed Northwind data model in Client SF

other columns show how many objects were transformed into the type denoted by that column.

We can see that the data-model of Northwind, consisting of `tables`, `columns`, `dom`, `foreign_keys` and `referential_constraints` objects, is transformed entirely since there are no untransformed objects left. For all these objects we can see how many objects are transformed into what type of TSF object. Notice further that the `Module` objects are shown in the cube as untransformed. This is because the cube ignores local transformations due to pre-processors, even though there exist trace links from the `Module` to the `Routine` objects. Furthermore, the `Routine` objects are transformed into 10 Application logic specifications and also into 6 static assignments (which connect an application logic specification to a table). We can see that the `Routine` objects are not transformed entirely; there are 45 untransformed `Routine` objects left. This is due to the fact that we limited the transformation of the Routines to the AfterUpdate eventhandlers. For all the untransformed objects, a proxy object is created in the TSF to remind the developer of these objects.

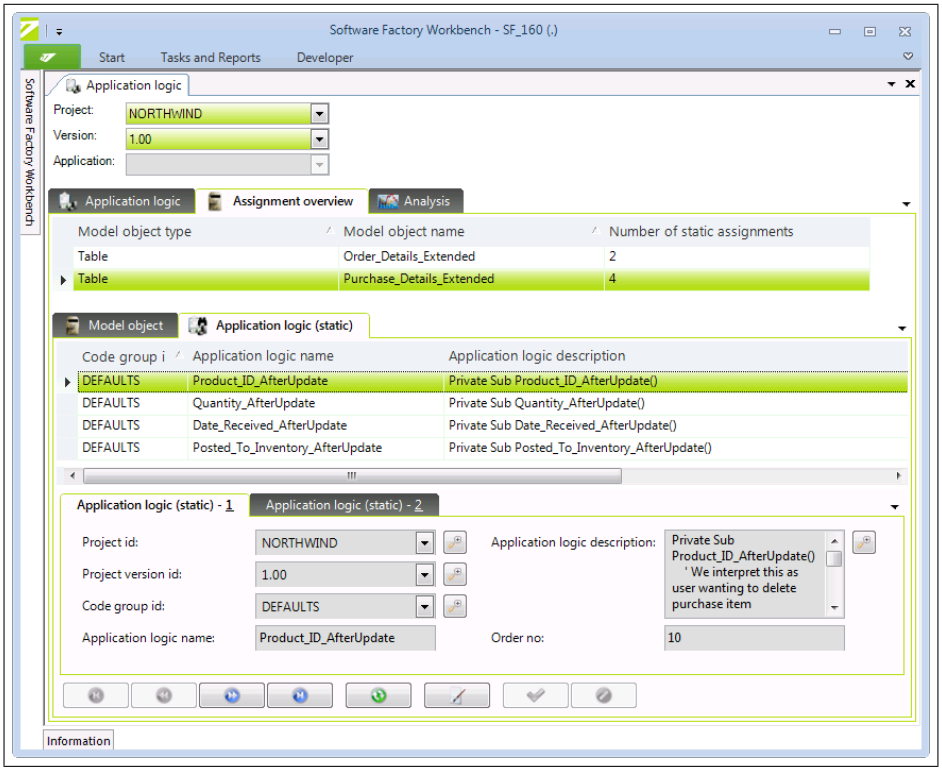

Figure 7.12: Transformed application logic specifications for Northwind in Client SF

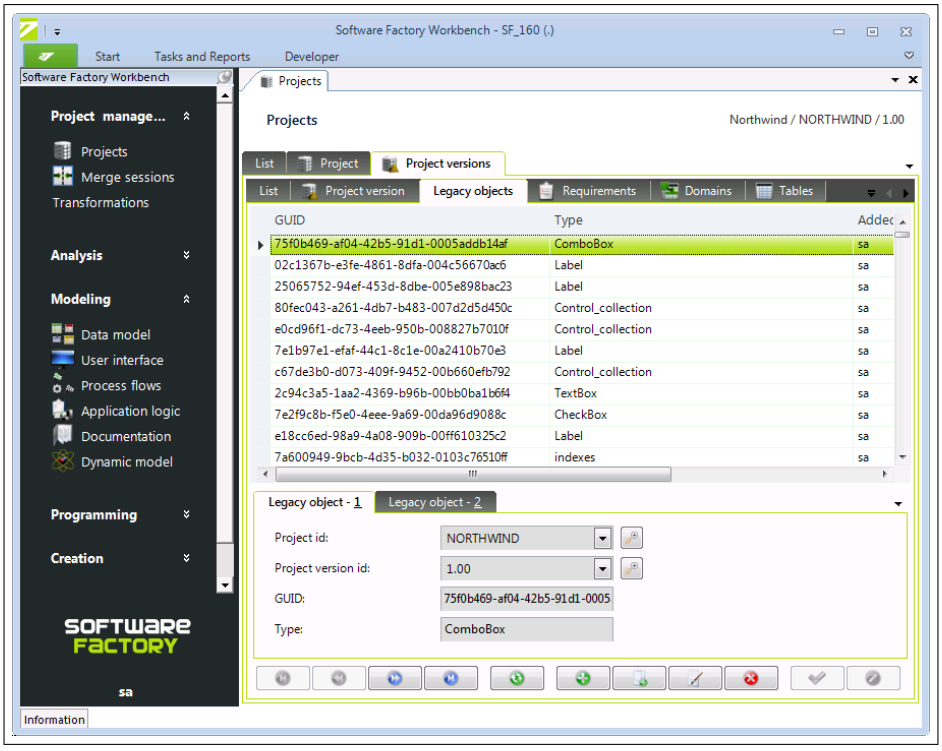

Figure 7.13: Untransformed proxy objects in Client SF

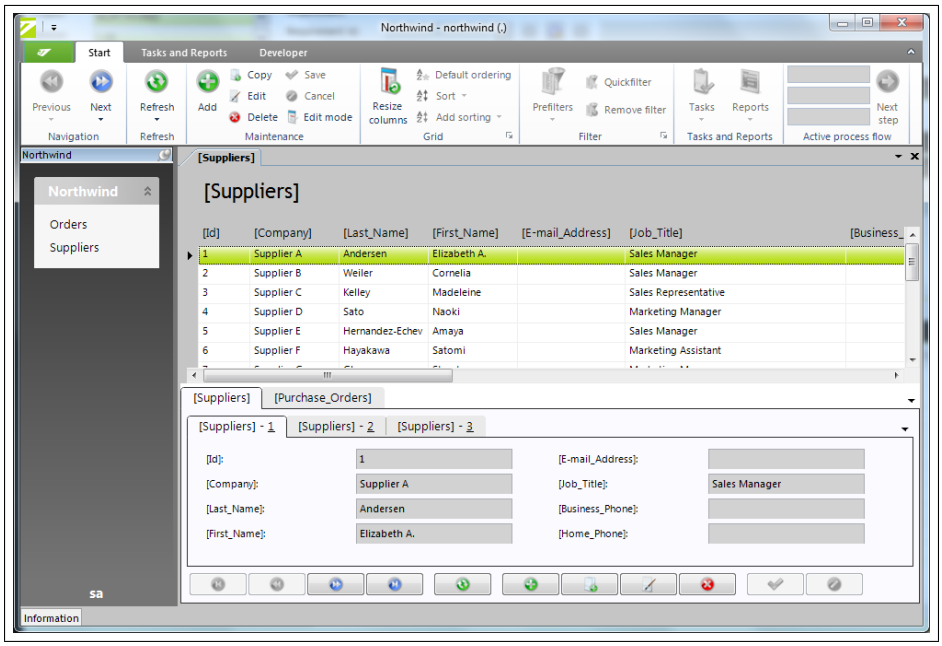

Figure 7.14: End Product

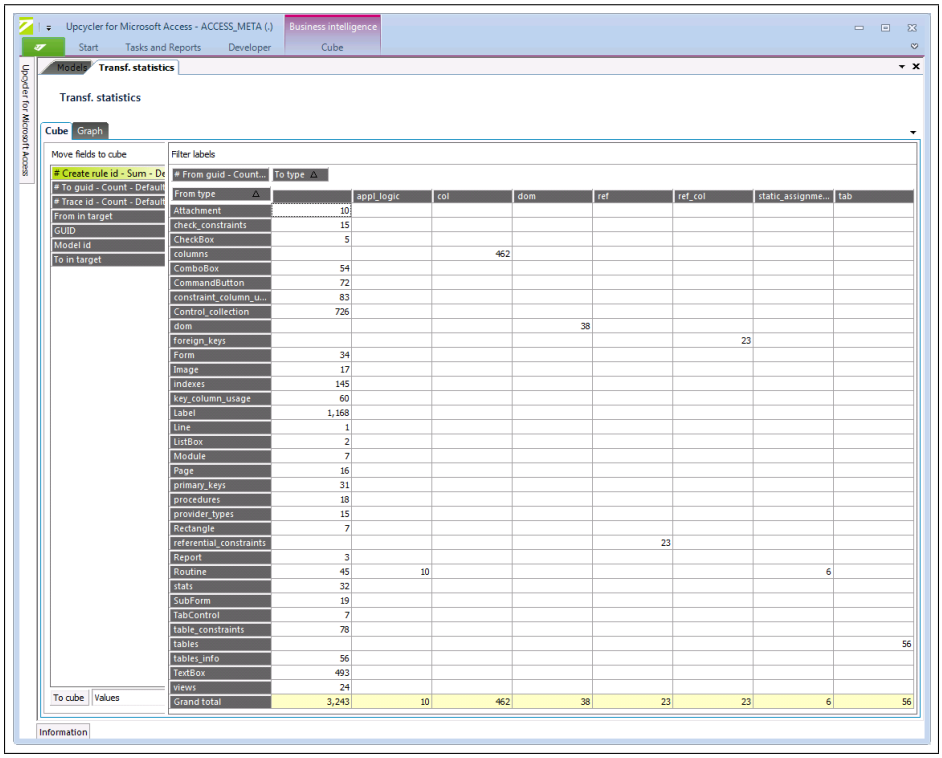

Figure 7.15: Business intelligence cube for transformation statistics

7.5 Conclusions

In this chapter we discussed the validation of our method to modernize legacy information systems. The subject of our pilot modernization project was the Northwind Access application. In order to modernize this application with the proposed method, we first needed an implementation of the method for Access applications: the Access Upcycler. To create the Access Upcycler – and Upcyclers in general – the TSF was extended with a framework that supported its development.

The Access Upcyler was used to extract the Northwind model from Access, using Microsoft Interop functionality. After this, the Routines pre-processor was executed to split the VBA Modules into individual Function and Sub Routines. The resulting model was then transformed and exported to the TSF with the transformation code generated from the transformation model. The transformation model for the pilot project consisted of mappings for the data-model, and for the Routines. Subsequently, the TSF was used to generate a new end product, in which (part of) the Northwind data was copied from the original database. After opening the new application and visualizing the same data, the overall method seemed successful. However, for a precise evaluation, we have to look if the stated criteria from chapter 1 are validated with this pilot project.

In the introduction we already formulated the following criteria which were to be validated in this chapter:

1. Best-effort approach
2. Not only data-model, but also other models can be transformed
3. Automatic traceability
4. Easy to use for someone with experience with the TSF
5. Allows for the use of custom code
6. Extensible for new legacy systems

The use of a best-effort approach becomes clear in the transformation of the VBA routines. Firstly, because the routines that are transformed are not transformed completely into working business rules in the target system. Instead, they are in an intermediate form (application logic specifications) and need to be manually translated further. Secondly, the majority of the routines remain untransformed. Recall that for the completeness, proxy objects are created in the TSF for these untransformed objects. This partial transformation would not be possible in an all-or-nothing approach, and therefore demonstrates a best-effort approach.

From the transformed data-model, and the end result we can see The same transformation of VBA routines shows us that our method is not limited to data-models only, since this is a transformation of business logic.

The automatic traceability is validated with the generated trace links. With these trace links, every object that is created due to a pre-processor or a transformation rule can be traced back to its source object. The statistics of the transformations in our pilot project were shown in a business intelligence cube.

Using this cube, we were able to determine that the data-model was transformed completely, and that the routines were transformed partially.

The fact that the Upcyclers are created with the TSF, makes the development (apart from the new transformation model) of Upcyclers very similar to that of other applications. Furthermore, by using the TSF, we also employ the same abstract TSF GUI's for all the Upcyclers. This validates the point that upcycling should be easy to do for someone with experience with the TSF.

The requirement that the method should allow for the use of custom code, was validated by the routines pre-processor. This pre-processor consisted of custom code to analyse the VBA modules and extract the individual Function and Sub routines contained in them.

Finally, the overall design of the prototype serves as the validation for the requirement that the prototype is extensible to support new legacy systems. For instance, if we want to upcycle an application of another type than that of Access, a new Upcycler can be created for that type of legacy system. The development of the new Upcycler, is supported by the framework. After this, the newly created Upcycler can be used for modernizing the application, but it can also be used for all other applications of that type. If, on the other hand, we want to upcycle another Access application, we can reuse the Access Upcycler created earlier.

Chapter 8

Conclusions and future work

8.1 Conclusions

In this thesis, we developed and validated a method for modernizing legacy information systems using Model Driven Engineering (MDE). The method bridges the gap between a legacy system and MDE, by transforming the legacy system into a model driven system. The model driven system can then be maintained and developed further with a specific MDE tool. Furthermore, the transformation of the legacy system itself is also accomplished with the MDE tool. This makes the used MDE tool not only the target environment of the transformation, but also the enabler of it.

An important choice in the development of the method, was to divide the modernization into two steps. We identified that, in order to modernize a legacy system, both the technical space and the meta-model (or the language / concepts) used by the legacy system need to change. In the first step, only the technical space is changed by extracting a model from the legacy system and store it in the technical space of the MDE tool. This model does not have to be compatible with the MDE tool, and can have its own meta-model. Since no model transformation takes place in this first step, it is relatively easy to perform, and as such, it also minimizes loss. In the second step of the method, the meta-model is changed to that of the MDE tool, by performing a model transformation. Because, at this point, the (legacy) model is already in the technical space of the MDE tool, this transformation can be carried out with the MDE tool itself.

A prominent difficulty in developing a general method for modernizing legacy systems, is that there are many different types of legacy systems. As such, the general method needs to be flexible with respect to the type of legacy system that needs to be modernized. This flexibility is realized in the method, by creating a meta-model for each type of legacy system. We have shown that this is more effective than a meta-model that generalizes all legacy systems,

because the former allows for a better use of model-generic functionality. Since model-generic functionality enables the reuse of transformation logic, this is important. Moreover, it will depend on the type of legacy system, how successful the meta-model will actually be in both generalizing the systems of that type, and allowing for model generic functionality at the same time. With highly structured legacy systems, like Microsoft Access, we have shown that this can be quite successful. But with lesser structured systems, it is likely that the model generic functionality is less effective, or less systems can be generalized with a single meta-model.

We addressed the model transformation problem by reducing it to a schema mapping problem. This can be done by representing meta-models by (relational) data schemes, and models by the data in those schemes. To express the mapping of the schemas, a transformation meta-model was developed. This meta-model allows one to specify (one to one) mappings of entities, attributes and relations between entities – thereby creating a transformation model. Furthermore, the meta-model ensures that the code to execute the transformation, and that to create the trace links, can be generated deterministically from the transformation model. This means that the developer only has to define a transformation model to get executable transformation code, which is traceable as well. When more elaborate (than one to one) transformations are needed, custom made pre-processors are used to transform the source model in such a way that, after this, the regular transformation model can be used. In this way, the transformation meta-model offers a practical basis to define model transformations, but can be enhanced with custom pre-processors when more flexibility is required.

The method was validated with a pilot modernization project, using the Thinkwise Software Factory as the MDE tool. To accomplish this, the TSF was extended with the transformation meta-model and with a framework to generate the transformation logic and trace links. The framework can be used to create a modernization application for each type of legacy system. We call these applications: Upcyclers. For the pilot project, the Northwind sample application for Microsoft Access was used. Therefore, an Access Upcycler was created to accomplish the modernization of this application. The pilot project validated the presence of the most important qualities of the method, including: a best-effort approach, not restricted to data-models, and automatic traceability. Furthermore, the requirements on the prototype were also validated, which entailed that: upcycling is easy to do for experienced TSF developers, custom code can be used in the transformation, and that the prototype is extensible to support other legacy systems in the future.

Overall, the proposed method in this thesis provides a uniform approach to modernizing legacy systems. While the general approach is equal in all modernization projects, specific functionality and transformation logic is created for each type of legacy system. This means there is a uniform basis, without sacrificing flexibility. The method aims to transform as much as possible from a legacy system, which is not restricted to only data-models. The best-effort character of the method allows for partial transformations, and all transformations are traceable. The use of MDE in modernizing legacy systems, proved to be useful as well, which allows for a more abstract definition of model trans-

formations. Furthermore, MDE, in combination with meta-modelling, allows for much reuse of functionality. For every legacy system of the same type, the pre-processors and transformation logic is reused. The framework, in which the general approach is implemented, can even be reused in every modernization project.

8.2 Future work

The work in this thesis can be seen as a first step in modernizing legacy information systems with Model Driven Engineering. As such, it provides several opportunities for improvement and enhancement in future work. In this section, we discuss these opportunities.

In the first step of the general approach, where the source model is created, we briefly discussed the use of reverse engineering. However, for our pilot project, no advanced reverse engineering techniques were required. This was because Microsoft Access allowed for easy extraction through the Interop API. But this can be different for other systems. Therefore, an investigation of which reverse engineering techniques are effective for which legacy systems would be useful.

In the proposed transformation meta-model, a transformation is built up from a set of transformation rules. When a transformation is executed to modernize a legacy system, it always entails the execution of all its associated transformation rules. We can imagine, however, that we sometimes want to limit the execution of a transformation slightly, by leaving out certain rules. It would therefore be a welcome enhancement if we could choose, at execution time, which rules need to be used in a particular transformation. This choice at execution time, can be further utilized by extending the transformation meta-model to enable the creation of multiple alternative rules. From a set of alternative transformation rules, only one can be chosen at execution time.

Even more flexibility regarding the execution of the transformation rules is required, when company specific naming conventions need to be taken into account in a transformation. In many legacy systems, naming conventions are used to impose additional structure on the construction of the system. When these naming conventions are known, they can make the transformation more effective. However, since these are *conventions* and not strict rules, inherently, they can be different at every company. Therefore, they cannot be put in the legacy meta-model (since that would assume all companies would use the same conventions). Instead, naming conventions (and potentially other company specific knowledge) need to be stored next to the legacy model, at the model level. It would be an interesting challenge to construct a useful meta-model for modelling conventions, and also to determine how we can influence the transformation with this extra knowledge.

Furthermore, the fact that MDE is used for the transformation can be utilized more as well. For instance, we could perform analysis on the transformation model, to validate it for completeness or to detect conflicts between transformation rules. These analyses might be useful at both creation and executing

time. At execution time, the input of the transformation (the source model) could possibly also be used in the analyses.

Because in most cases a legacy system cannot be transformed entirely, we employed a best-effort strategy. This means that elements that can be transformed are transformed, and elements that cannot be transformed are left as-is. This is clear for elements for which we can determine the transform-ability with certainty. In practice, however, it might not always be so clear if and how an element can be transformed – which introduces an amount of uncertainty. Furthermore, uncertainty can also occur in the results of the more elaborate reverse engineering techniques discussed above. It would be interesting if we could make our "best"-effort even better by incorporating these uncertainties as well. A possible approach to address this uncertainty, is to employ a probabilistic database for the storage of the source, the target and the transformation model. In a probabilistic database, all the possibilities are stored along with their associated probability [26].

Finally, the thesis focussed only on the transformation of a legacy system into a model driven system and not on general improvements to further modernize the legacy system. An investigation of general improvements – to be carried out in the MDE tool, after the transformation – would therefore be interesting. This would literally pick up the system where we left off, and continue with the challenge of modernizing legacy information systems with MDE.

Bibliography

[1] W. Ulrich, *Legacy systems: transformation strategies*. Yourdon Press Computing Series, Prentice Hall, 2002.

[2] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: issues and directions," *Software, IEEE*, vol. 16, pp. 103 –111, sep/oct 1999.

[3] Wikipedia, "Wikipedia: Legacy system." `http://en.wikipedia.org/wiki/Legacy_system`, July 2011.

[4] S. Kent, "Model driven engineering," in *Integrated Formal Methods* (M. Butler, L. Petre, and K. Sere, eds.), vol. 2335 of *Lecture Notes in Computer Science*, pp. 286–298, Springer Berlin / Heidelberg, 2002.

[5] M. J. Miller, J., "Mda guide version 1.0.1.." OMG report, 2003.

[6] O. M. Group, "Object management group: Model driven architecture." `http://www.omg.org/mda`, August 2011.

[7] T. Mens and P. V. Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[8] F. Jouault, "Loosely coupled traceability for atl," in *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pp. 29–37, 2005.

[9] M. Brodie and M. Stonebraker, *Migrating legacy systems: gateways, interfaces & the incremental approach*. The Morgan Kaufmann series in data management systems, Morgan Kaufmann Publishers, 1995.

[10] O. M. Group, "Object management group." `http://www.omg.org`, August 2011.

[11] O. M. Group, "Object management group: Meta object facility." `http://www.omg.org/mof`, August 2011.

[12] Wikipedia, "Wikipedia: Meta object facility." `http://en.wikipedia.org/wiki/Meta-Object_Facility`, August 2011.

[13] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference* (J.-M. Bruel, ed.), vol. 3844 of

Lecture Notes in Computer Science, pp. 128–138, Springer Berlin / Heidelberg, 2006.

[14] J.-L. Hainaut, *Introduction to database reverse engineering*. University of Namur, 2002. Technical report.

[15] Eclipse, "Eclipse: Modisco." `http://eclipse.org/MoDisco`, August 2011.

[16] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[17] O. M. Group, "Object management group: Architecture driven modernization." `http://adm.omg.org`, August 2011.

[18] O. M. Group, "Object management group: Knowledge discovery metamodel." `http://www.omg.org/spec/KDM/1.1`, August 2011.

[19] K. Analytics, "Kdm analytics: Knowledge discovery meta-model." `http://kdmanalytics.com/kdm`, August 2011.

[20] P. Atzeni, P. Cappellari, and P. A. Bernstein, "Modelgen: Model independent schema translation," *Data Engineering, International Conference on*, vol. 0, pp. 1111–1112, 2005.

[21] P. Atzeni, P. Cappellari, and P. Bernstein, "A multilevel dictionary for model management," in *Conceptual Modeling ER 2005* (L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, and O. Pastor, eds.), vol. 3716 of *Lecture Notes in Computer Science*, pp. 160–175, Springer Berlin / Heidelberg, 2005.

[22] J. Bezivin and I. Kurtev, "Model-based technology integration with the technical space concept," in *In: Proceedings of the Metainformatics Symposium, Springer-Verlag*, Springer-Verlag, 2005.

[23] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," 2003.

[24] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the 27th international conference on Software engineering*, ICSE '05, (New York, NY, USA), pp. 49–58, ACM, 2005.

[25] Microsoft, "Interoperability (c# programming guide)." `http://msdn.microsoft.com/en-us/library/ms173184.aspx`, September 2011.

[26] A. de Keijzer and M. van Keulen, "A possible world approach to uncertain relational data," *Database and Expert Systems Applications, International Workshop on*, vol. 0, pp. 922–926, 2004.