A new approach to managing data model and database co-evolution

Master's Thesis

Phil Hunte

A new approach to managing data model and database co-evolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Phil Hunte
born in Willemstad, Curacao

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Avanade Netherlands BV
Versterkerstraat 6
1322 AP
Almere, the Netherlands
www.avanade.nl

A new approach to managing data model and database co-evolution

Author: Phil Hunte
Student id: 1328557
Email: p.hunte@student.tudelft.nl

Abstract

Applications evolve over the course of time, therefore their data models need to be adapted. To reduce the effort required for data model and database co-evolution, an automation solution is needed that can transform a database in such a way that it can store the new data whilst also retaining the existing data.

This thesis presents a redesign of the operation-based approach to automating the coupled evolution of graphical data models and their data. In addition to its automated co-evolution support, the redesign also addresses the inherent limitations of the operation-based approach.

On the basis of an exploratory evaluation it was concluded that the proposed approach indeed supports the co-evolution of graphical data models and that it results in an increase of the expressivity and the adaptability of the approach.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. P. Pawelczak, Faculty EEMCS, TU Delft
University supervisor:	Dr. G. Gousios, Faculty EEMCS, TU Delft
Company supervisors:	MSc. S. Bockting, Avanade Netherlands BV
	MSc. G. Ramaker, Avanade Netherlands BV

Preface

"The roots of education are bitter, but the fruit is sweet"
– Aristotle

The purpose of this research project is to redesign an operation-based approach to automate the co-evolution of graphical data models and the database that they represent. The redesign focuses on the design of the internal components and on the mitigation of the inherent limitation of operation-based approaches.

The research project consisted of a literature survey and a design research. The literature survey and the first half of the thesis research were carried out at Avanade from April 2012 to February 2013. Avanade is a joint venture between Accenture (a management consulting, technology services and outsourcing company) and Microsoft. Avanade was chosen as the location for this research because of its expertise in Microsoft technologies, which meant that they could provide the necessary tooling and expert support needed for a successful completion of the research project.

March of 2012 marked the beginning of this thesis project process. It began with three months of searching for a TU supervisor and three research topic changes, after which I received the go-ahead for conducting a literature survey on automating model co-evolution and database transformation. After having concluded the literature survey it took another three months of research scope changes to define a thesis proposal. Each change of the thesis scope was accompanied by moments of frustration, self-doubt, and tiredness. This uncertainty was carried into the project until April of 2013 when the evaluation of the research lead to the insight behind the current project objective. As a consequence, the majority of the thesis was restructured.

This report is the result of the restructured design research project done on automating the co-evolution of graphical data models.

Phil Hunte
Delft, the Netherlands
October 22, 2013

Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge those who have assisted and supported me with the following words of gratitude. To Sander Bockting, thanks for the unwavering optimism. To Gijs Ramaker, thanks for the insights on planning and presentation skills. A thanks to both of you for the given space during the process of changing my thesis topic and finding a TU supervisor.

To Georgios Gousios, thanks for the candid feedback and the brainstorming session-s/meetings early on. To Andy Zaidman, thanks for challenging me to make my writing more explicit and to keep moving forward. A thanks to both of you for putting up with my drive to perfect that which was considered to be 'good enough'.

In closing, thanks to all my supervisors for their provided insights, effort, time, and given space which allowed me to work at my own pace. Consequently, I could work-out multiple ideas until I found the one that best fit my research project. And last but not least, thanks to my friends and family who expressed genuine interest in how I was doing and to Tanya for proof reading sections of this report even though they were too technical for a layperson (e.g. "Hey, what does SQL mean?" ◡̈).

Chapter 1

Introduction

Software applications are continuously evolving. This evolution is evident throughout the entire development life cycle, from initial development to maintenance, and includes the introduction of new features, the improvement of old features, and bug fixing [16]. Applications developed using the Model-Driven Engineering approach are not immune to this. Model-Driven Engineering (MDE) [65] is a software engineering paradigm that aims to facilitate software development by creating, maintaining and manipulating abstractions which represent the system. These abstractions, also known as models, enable developers to create and evolve software at a higher level of abstraction which, in turn, simplifies the development effort [27].

One aspect that needs to be taken into account when a model of a software system has evolved is the propagation of this change to the vital components which are not directly represented in the model. Cicchetti et al. [9] term this problem; the maintenance of overall application validity. They state that the validity of Model-Driven (MD) applications is based on interdependencies between the meta-model and model, as well as between the model and the aspects that are not depicted in it (i.e. the non-reflected aspects such as data and manual customization). Therefore, to ensure the validity of a system, the (meta-)model changes need to be propagated to both the reflective and the non-reflected aspects. Furthermore, Cicchetti et al. state that the more formally these non-reflected aspects are defined, the easier it becomes to automate the management of the system's validity.

1.1 Project Motivation

This research project focuses on the system validity management issue that occurs when dealing with the evolution of database-centric applications developed using the MDE approach. Such applications generally rely on a data model to store application data into a database [73]. Therefore, as these applications evolve so do their data models. An example of this is the changing of an application's logical structure. This would require a new database schema which, in turn, leads to changes in the data model to accommodate the new manner in which the application data should be stored [1]. Ideally the application should automatically maintain its validity by migrating its database (along with its data) to con-

form to the modified application structure. However, the deployment of a new application version is often accompanied by the creation of a new database [70]. This is a straightforward approach to ensure that the application uses a database that conforms to the new data model. Consequently, the original data in the existing database becomes inaccessible to the new application. To resolve this issue the existing data needs to be migrated to the new database. Since manual migration of database tables is tedious and error-prone, adequate tooling is needed to support the necessary data migration [10].

1.2 Problem Definition

There have been a number of advances in model evolution automation which could be used to reduce the human effort needed to support the coupled evolution (co-evolution) of data models and the database that they represent [8, 23, 64, 36, 74]. The current state-of-the-art model evolution approaches focus on the reconstruction of the evolution steps using model matching techniques [64]. The result of this reconstruction is a set of identified model differences. These model differences can then be used to co-evolve the database. This two-step process could be used to automate the co-evolution of a data model and its database. However, in the case of data models with a graphical notation this model matching approach has a distinct drawback. Identifying the changes made to a graphical model by comparing model versions is a complex operation, because it can be reduced to the graph isomorphism problem which is NP-hard [61]. The use of model matching techniques, therefore, do not result in an exact identification of the occurred model evolution; but rather an approximation thereof. As a consequence of this approximation, corrective user input becomes necessary.

According to the literature survey [41] conducted prior to this research, there is another possible approach to automating data model and database conformance which is to use an operation-based approach instead of the model matching approach. The literature survey[1] concluded that operation-based co-evolution was the preferred approach for coupling model transformations to data migration. This is because it tracks model adaptations more accurately and it provides a mechanism to map model adaptations to arbitrary transformations. The latter can be used to map model adaptations to corresponding database transformations. Consequently, the operation-based approach seems to be the better choice, but it too has inherent limitations.

The operation-based approach uses operators to execute co-evolution tasks [45]. If the existing operators cannot realize the required task, then there are two outcomes. Either the approach is discarded in favor of another one that does support the needed task, or a new operator is created to handle it. Thus, if the approach relies solely on its existing set of operators, it is not expressive enough to fully support custom co-evolution tasks. In other words, it should be possible to define new operators. Some implementations of the operation-based approach use general-purpose languages to define their operators. This allows for enough coding flexibility. However, these languages do not provide a means to capture migration patterns or other reoccurring patterns [31]. Therefore, the creation of a new operator involves writing both the model transformation statements and the corre-

[1] The details of this literature survey are outlined in Chapter 2

sponding database transformation statements. As mentioned before, this process is tedious and error-prone. Furthermore, scripts are used to evolve an existing database. Such a script contains the set of Structured Query Language (SQL) statements needed to migrate the database. In an operation-based approach the script would be generated from the recorded model-to-database mapping output. If, for any reason, any of the script's statements need to be changed, there are two options: the first is to change the script manually (which would require one to be able to find the statements that need to be changed), the second would be to undo the relevant model transformations and then re-apply them in order to update the migration script. Either option would be tedious and error-prone.

Though the operation-based approach seems to be the better choice for automating the co-evolution of data models, its inherent limitations need to be addressed in order for it to provide a more extensible and adaptable automation support.

1.3 Project Goal

The goal of this research project is to find a way to redesign the operation-based approach so that it could provide better support for the coupled evolution of graphical data models and their data. Therefore, a redesign was proposed that aims to improve the approach by doing two things. The first is designing a new operator definition framework that addresses the operator creation limitations and thereby facilitate extensibility. The second is to redesign the way in which model modifications are recorded in order to facilitate the adaptation of the generated database transformations, without it having to affect the applied model transformations.

1.4 Research Method

Based on the motivation and goal for this research, the following research question was formulated: *"How can the limitations inherent to the operation-based approach to automating the co-evolution of data models with graphical notation be mitigated by using operator composition and event-sourcing patterns?"*

In order to systematically tackle the research question, it was divided into the following sub-questions:

- *RQ1* - What are the limitations inherent to the operation-based approach to automating the co-evolution of data models with a graphical notation?
- *RQ2* - How can the use of operator composition reduce the effort required to define new operators?
- *RQ3* - How can the event-sourcing pattern be used to manage the process required to map data model adaptations to a database transformation?

1.5 Thesis Structure

This thesis report consists of six chapters which are structured as follows. Chapter 2 provides the research background and related work. Chapter 3 presents the proposed solution. The techniques used to realize a prototype of the proposed redesign are described in Chapter 4. This is then followed by the evaluation of the redesign in Chapter 5 and Chapter 6 concludes the report.

Chapter 2

Background

This chapter introduces the concepts and fields of research that form the basis to the thesis. Section 2.1 introduces the terminology and principles that pertain to the coupled evolution of data models and their data. Section 2.2 reviews the three main approaches used for automating coupled evolution. Furthermore, an overview of the available tools is presented in Section 2.3. Finally, the research areas relating to coupling data models and databases are described in Section 2.4.

2.1 Introduction to Coupled Evolution

Lämmel states that coupled evolution is defined as a process in which mutually dependent software artifacts of different kinds are simultaneously transformed. In addition, the transformation is centered on a grammar (or schema, Application Programming Interface (API), or a similar structure) that is shared between the artifacts [50] [49]. Simply put, coupled evolution can be defined as a development activity in which several artifacts are all changed to maintain consistency. Co-evolution can be performed by first changing a single artifact (e.g. source code) after which each dependent artifact is changed to re-establish consistency [64]; the latter activity will be called migration.

2.1.1 Terminology

The concepts that will be discussed throughout this report may contain ambiguous terms. For that reason, this section provides a concise description of the main concepts.

Model: A model is an abstraction of an object. In the case of this thesis, the object instance is a database containing application data. Models are represented through a model language - also termed Domain-Specific Language (DSL) - which encapsulates the concepts, relationships and constraints pertaining to the specific application domain [16]. The DSL uses a notation that conforms to the syntax and semantics of the domain, thus making it easier and more effective to develop with than general-purpose programming languages [54]. This notation can be graphical or textual [65] (cf. Figure 2.1 and Listing 2.1).

A particular advantage of a graphical DSL is its use of a visual interface that allows someone who is a domain expert but not a software developer to use it as an application development tool. A graphical DSL is often a restrictive language that only consists of domain concepts and it has well-formedness rules that constrain the domain programs that can be created with it [67].

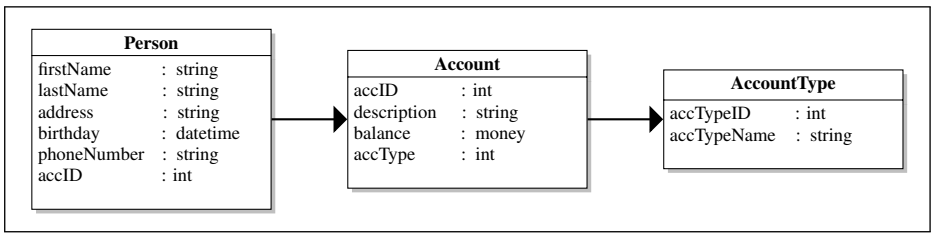

Figure 2.1: Visual Data Model

```
1  class Person {
2      string    firstName;
3      string    lastName;
4      string    address;
5      datetime  birthdate;
6      string    phoneNumber;
7      Account   accID;
8  }
9
10 class Account {
11     string      accID;
12     string      description;
13     string      balance;
14     AccountType accTypeID;
15 }
16
17 class AccountType {
18     int     accTypeID
19     string  accTypeName
20 }
```

Listing 2.1: Equivalent Textual Data Model

Data Model: Data models are an aspect of software development. They define the structure of data that is processed by an application and the schema of a database. Applications produce and store data that conforms to the data model [72].

A data model consists of entity declarations which have a name and a set of properties. Each property has a name and a type. An entity declaration might inherit from another entity declaration which means that there can be a hierarchical structure.

Model Evolution: The collective term for the coupled evolution of meta-models and their model instances is model evolution. Another variation of this collective term is coupled model evolution or model migration.

Model Transformation: Model transformation is a development activity in which a model is adapted according to certain well-defined specifications. There are three types of model transformations: there are model-to-model transformation, model-to-text-transformation and text-to-model transformation [14]. Each type of transformation has unique characteristics and tools, yet they all share some common characteristics. This thesis focuses on the model-to-model transformation type.

Data Model Evolution or Data Model Co-Evolution: This is the collective term for the migration of a database as a result of changes to its data model.

Coupled Operator: A coupled operator is a set of model transformation statements and corresponding database transformation statements. Therefore, when an operator is applied to a data model it transforms it according to its set of model transformation statements and it produces the SQL code necessary to migrate the existing database. The execution of the produced SQL statements will reestablish the conformance relation between the new data model and the existing database.

2.1.2 The Coupled Evolution Problem

Lämmel gives a two-part definition for the "coupled evolution problem". The first part states that any instance of coupled evolution must define the notion of consistency for the involved artifacts. Meaning, a coupled evolution starts from and finishes with a consistent collection of artifacts. Secondly, any kind of coupled evolution instance must also define the notion of migration, which defines how evolutions of one artifact affect all the other artifacts [49]. In other words, consider without loss of generality the re-establishment of consistency (or reconciliation) of two artifacts. Let A and B be these artifacts. We assume a consistency relation c on A and B, and we are given two concrete artifacts a:A and b:B such that *c(a, b)* holds. We consider a type-preserving transformation on A, denoted by g, and we apply this transformation to a such that we obtain $a = g(a)$. Then, the reconciliation is about determining a suitable b such that *c(a, b)* holds.

In the same work, Lämmel states that there are a number of reconciliation options, such as *no reconciliation*, *degenerated reconciliation*, *symmetric reconciliation*, *asymmetric reconciliation* and *reconciliation by matching*. Figure 2.2 depicts the *asymmetric reconciliation* option. The continuous arrows show the transformations and dashed arrows represent consistency claims.

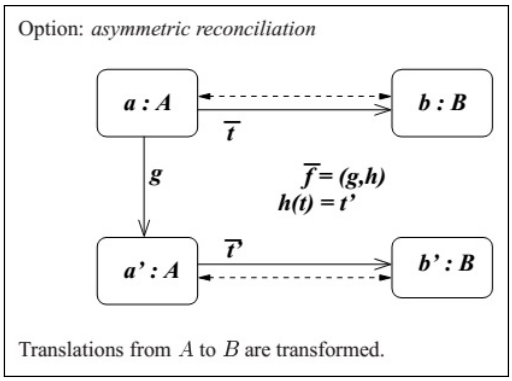

Figure 2.2: Artifact Migration Option - Asymmetric Reconciliation

Asymmetric reconciliation is based on the assumption that there is access to an actual translation of a to b. It requires a description t of this translation in terms of a dedicated transformation language. Then, the actual translation from a to b is the interpretation $\bar{t}$ of the description t. The f is a transformation description which is used to maintain a link between two artifacts. The interpretation of the description f, denoted as $\bar{f}$, provides two actual transformations, one on A, and another on B. In other words, if A is the data model and B is the database that it represents, then t is the coupled operator that is responsible for transforming the two. The set of transformations within the operator that will transform both A and B is $\bar{f}$.

This option involves two critical issues. First, one must identify a language for defining $\bar{f}$. Second, the primary transformation language (i.e. the model transformation language) must be defined such that its effect can also be properly mapped to the data transformation language [49].

2.1.3 Coupled Evolution Domains

Coupled evolution problems are ubiquitous; they are encountered in various domains of computer science, e.g. in language processing, generative programming, automated software engineering, software re-engineering, model-driven architecture, and database re-engineering [49]. This section describes the coupled evolution domains that are relevant to model evolution and database evolution.

Coupled Evolution of Models

The co-evolution of models deals with the maintenance of the consistency between a model and its meta-model. Meta-models describe the set of available domain concepts and constraints, i.e. the syntax [6], that a proper model must conform to [43]. Therefore, when a meta-model is adapted, existing models may no longer conform to the adapted meta-model. These models have to be migrated so that they can be used with the evolved modeling language. Manually migrating existing models to the adapted meta-model is tedious and

error-prone, hence the need for tool support which reduce the effort associated with model migration.

Coupled evolution for the meta-model domain was introduced by Gruschko [24]. As is the case for most publications on coupled evolution of models, he models evolution using small elementary transformation steps. Wachsmuth applies coupled evolution to Meta Object Facility (MOF) [1] compliant meta-models [74, 39]. MOF is an extensible model driven integration framework for defining, manipulating and integrating meta-data and data in a platform independent manner. Ecore, the meta-meta model used in the model development environment Eclipse Modeling Framework (EMF) [21], is a variant on MOF. Wachsmuth introduced a set of transformations and proposes a mapping to model migrations implemented in the Query-View-Transformation (QVT) model transformation language [58]. Similar to Wachsmuth, Herrmannsdörfer looked at coupled model evolution based on small evolution steps [30]. He focused on EMF itself.

Coupled Evolution of Databases

Analogous to the model and meta-model coupled evolution, there is the (database) schema evolution in which the aim is to maintain the consistency between a schema and the database instance that it pertains to. This type of coupled evolution can be seen as a variation of the *data model evolution* [69].

The main difference between a meta-model and a database schema is that database schemas are a domain-specific modeling language for database development. However, they serve as a descriptor of meta-data, and are therefore subject to the same challenges when their meta-data changes. Furthermore, schemas are used to describe a set of concepts. The schemas provide an abstraction containing only those concepts which are relevant [18]. Therefore, schemas in these domains may be thought of as being analogous to meta-models because they provide a means for describing an abstraction over a real-world instance. Consequently, approaches to identifying, analyzing and performing schema evolution are relevant to the evolution of data models in MDE.

Many of the published works on schema evolution focus on defining a list of evolutionary operators which the developer is expected to apply to evolve schemas. This approach is used for Extensible Markup Language (XML) schema evolution [25, 68].

Lastly, data model evolution can also be seen as a variation of 'format evolution'. Format evolution is an instance of what Cunha et al. [13] refer to as two-level transformations. This is a transformation where a type-level transformation (the data type) determines or constrains value-level transformations (the data instances). Numerous approaches have been found to solve two-type evolution problems [51, 13, 5, 26, 3]. These mainly focus on the schema to data mapping for a specific type of schema.

[1] http://www.omg.org/spec/MOF/2.4.1/

2.2 Automating Coupled Evolution

Applying the various types of coupled evolution manually can by tedious and error-prone, therefore techniques are needed to automate aspects of the co-evolution process. Sections 2.2.1 and 2.2.2 give an overview of the main approach categories used to automate model evolution, namely: *difference-based* and *change-based* approaches.

2.2.1 Difference-based Co-Evolution

Difference-based approaches, also known as the model matching approach, try to automatically deduce model transformations from the difference between two snapshots of the model. There are various techniques available to identify model differences. Since this thesis is focused on models with a graphical notation, this section will only describe those techniques that fit within this scope.

Model Matching Techniques Algorithms are used to identify model differences. There are four types of algorithms used to match models with a graphical notation [47].

- *Static Identity-based Matching* With this type of algorithm each element has a persistent, non-volatile unique Id assigned upon creation. This Id is then used to match model elements to each other [2]. The advantages are that there is almost no initial setup cost; meaning there is no configuration requirement for such algorithms. And secondly, this technique tends to be quite fast compared to the remaining algorithm types. On the other hand, its drawbacks are that it does not apply to models constructed independently of each other, or model representations that do not support unique Ids.

- *Signature-based Matching* Much like the previous algorithm, this matching technique uses element Ids. The difference is that each model element Id is a signature calculated dynamically using a user defined function which is based on the elements features [62]. The benefit of using signatures is that it eliminates the shortcomings of the Static Identity-based algorithms. The disadvantage is that it requires configuration. The Id-signature calculation function needs to be defined for each model element type.

- *Similarity-based Matching* This algorithm attempts to identify matching model elements based on the aggregated similarity of their features. Each feature is assigned a relative weight. This weight indicates the importance of the item [55] [23] [19]. The advantage compared to the identity-based matching is that typed attribute graph matching algorithms have been shown to produce more accurate results. The drawbacks are that defining and fine-tuning the element weight is mostly a trial-and-error process, and because of its generic natures, it fails to take into account the semantics of the modeling language used. If this disadvantage is addressed one can reduce the search space which in turn increases the accuracy and the run-time performance of the algorithm.

- *Custom Language-specific Matching* This is a generic category for custom techniques developed for a specific language and based on that language's syntax. Therefore it is equally or more accurate than the other matching algorithms, with the added benefit of a better run-time performance due to the much smaller search space involved [7] [46]. Its shortcoming is that it requires specification of the complete matching algorithm which is a substantial implementation requirement. Furthermore, unlike the other matching techniques, this one may require custom development-tools to facilitate its model matching implementation.

These four matching algorithms provide a generic approach to diagram-matching for architectural models and data models. Thus, they are applicable to any graphical modeling language. However, the computational complexity of these algorithms makes them less feasible for large applications. The algorithms have high complexities with regard to memory allocation and computation time. Furthermore, even though these techniques are fully-automatic, they have the disadvantage that the derived migration may not be the one intended by the developer. Its dependence on availability of stable model element identifiers. In the absence of stable unique identifiers, the direct comparison would not be capable of detecting element moves and would detect a deletion followed by an addition instead. Consequently, the developer has to manually correct and understand the derived migration.

2.2.2 Change-based Co-Evolution

The change recording approach records a list of primitive changes made to a model and this list can be normalized to reduce redundancies. However, normalizing the list may also remove useful information such as the exact order in which each change occurred. Some types of modifications such as meta-model element renaming can be more easily recognized using this approach [64]. Much like model differencing, this technique does not always lead to an unambiguous result. This is due to the fact that the order in which the changes are recorded affects the inference of the transformation. Hence, this approach requires the developer to continuously be aware of the way the changes will be interpreted.

One of the challenges of using change recording is that the granularity of the changes that can be tracked influences the inference process. In other words, deciding on which granularity to use is more of a trial-and-error approach than a calculated choice. Furthermore, this approach still has the same limitation as the model differencing approach. Meaning, this approach is fully-automated yet there is a chance that the output it produces may need manual correction.

Operation-based Co-Evolution

Operation-based co-evolution is a specific instance of the change-based approach. It is based on specifying the evolution using a library of operators. Each operator specifies an evolution step as a mapping from a model change to one or more changes at the same or different abstraction level. The more expressive (complete) the library is, the more useful this approach becomes. In case there is a change for which there is no operator in the library the developer must either use another approach or specify this operator manually

[8]. Operation-based matching allows for a finer-grained specification of the model modifications. In other words, besides the obvious item addition, deletion and update, it is also possible to map additional transformations to an operator.

This approach avoids the manual correction and the understanding of derived migrations problem by using incremental transformations which enable the capturing of the intention while performing (meta-)model adaptation. Consequently, the developer has to take the impact on models into account which may constrain (meta-)model adaptation, but it also may leads to a more systematic modeling process. Wachsmuth adopts ideas from grammar engineering and proposes a classification of meta-model changes based on instance preservation properties [74]. Based on these preservation properties, Wachsmuth defined a set of high-level coupled operations.

2.3 Coupled Evolution Tools

There has been a number of advances in tooling to support automated coupled evolution. They differ in the type of coupled evolution they support. Rose et al. present a classification for such tools based on the approach they utilize [64]. The categories are as follows:

- *Manual specification* approaches provide languages tailored for model migration to manually specify the migration. These approaches do not provide explicit support to ensure that the specified migration is semantics-preserving.
- *Matching* approaches try to detect a migration based on the matching between two different snapshots (versions) of an item. The main downside to matching approaches is that they are not able to detect a semantics-preserving migration.
- *Operation-based* approaches specify the needed migration as a sequence of coupled operations which encapsulate the model adaptation and the required migration. Because of this it is possible to preserve some aspects of the semantic [74, 34].

The next sections detail a list of tools according to this classification which support the coupled evolution of data-models and databases.

2.3.1 Manual Specification Tools

Any Integrated Development Environment (IDE) which allows hand-written general-purpose programming languages such as Java, C# and SQL can be used to specify data model evolutions. However, there are a few which facilitate the process of coupled evolution. There are model development environments such as the aforementioned Eclipse Modeling Framework (EMF) [2], for example, that provide an editor which provides a variety of functionality that support model transformations. Such modeling environments can also facilitate database migrations by sending commands to database servers. Many tools have been written as plug-ins to such platforms.

The following is a list of the relevant manual migration tools:

[2]EMF web-site: http://www.eclipse.org/emf/

Ecore2Ecore: is a manual specification migration tool that is part of the EMF. Migration is specified with a mapping model and hand-written Java code. Ecore2Ecore has been used in real-world projects, such as the Eclipse MDT UML2 project , to manage co-evolution [63].

Epsilon Flock: is a manual specification migration tool. Flock is a domain-specific transformation language tailored for model migration. In particular, Flock automatically copies from original to migrated model all model elements that have not been affected by meta-model evolution. Flock is built atop Epsilon [3] which is an extensible platform providing inter-operable programming languages for model-driven development.

Visual Studio Visualization and Modeling SDK [4] This Software Development Kit (SDK) which integrates tools and templates for building Domain-Specific Language designers, extending Unified Modeling Language (UML) and Layer designers into Visual Studio. This model development environment is the .NET equivalent to the Eclipse based environment, however it is not as elaborate as the EMF.

Graph Modeling Framework: Graphical Modeling Framework (GMF) [5] is a DSL framework which is used to develop graphical editors based on EMF and Graphical Editing Framework (GEF) [6]. EMF provides a modeling and code generation framework for Eclipse applications based on structured data models, and GEF provides technology to create rich graphical editors and views for the Eclipse Workbench User Interface (UI). The GMF is defined as the bridge between these two frameworks that combines their features to enable the graphical development of models. The advantages of this tool are that it has an extensible framework that allows editors to be extended through custom code and it also enables the option of creating a distribution site using the Eclipse Plug-in Development Environment (PDE) features which help with keeping client installations up-to-date.

MetaEdit+: MetaEdit+ [7] is a two-part commercial domain-specific modeling environment. There is a MetaEdit+ Workbench with which the modeling language is designed, and there is the MetaEdit+ Modeler that enables development with the designed language.

The workbench provides a meta-modeling language and tool suite for designing language concepts, properties, associate rules, symbols, generators and checking reports. It also has an extensive library of reusable modeling language components [57].

The modeler enables the development of a language through graphical diagrams, as matrices or as tables. It uses four components (diagram, matrix and table editor and a browser) which together offer full modeling tool support [56].

The advantages of this tool are the amount of advanced features that it supports. To name a few, there is support for full integration of the toolset with existing tool-chains

[3] http://www.eclipse.org/epsilon/

[4] VS VMSDK website: http://archive.msdn.microsoft.com/vsvmsdk

[5] GMF web-site: http://www.eclipse.org/gmf

[6] GEF web-site: http://www.eclipse.org/gef/

[7] MetaEdit+ web-site: http://www.metacase.com/

through APIs and there is built-in support for language evolution. Furthermore, the multi-concurrent user support enables developers on different platforms and at different locations to collaborate on a project. One drawback is the base-price of €150,00 for 1 academic license.

2.3.2 Matching-based Tools

Matching-based tools model the evolution through a set of differences between the original and evolved version of a model. The changes that constitute the evolution are not recorded as they occur, rather they are recorded afterwards. In other words, they register the effect of the evolution.

The state-of-the-art matching-based approach for identifying model differences is the similarity flooding algorithm [66]. The state-of-art implementation for identifying model differences is the combination of the EMF Compare tool and the ECL tool [9]. Cicchetti et al. have proposed a data model evolution approach based on the EMF Compare and ECL tools. It uses this model differencing technique to realize a migration approach capable of detecting the modifications a model underwent during its lifecycle after which it automatically derives from them the programs that are capable of migrating various aspects of the whole application including the data persistently stored in a database. They use Acceleo [8] to realize this data migration. The following is a list of matching-based tools such as Acceleo.

Acceleo: This is a template-based approach for generating text from models. It can be used to generate SQL queries to update a database schema and to migrate the existing stored data based on a difference model. Its output is a text-based script which requires a separate tool to execute [9].

Acoda: Acoda is a model matching approach for co-evolving object-oriented data models and their database. It provides a domain-specific language for specifying data model evolution as a separate concern at the data model level. The input are two versions of a data model and a database. Acoda the automatically detects what was changed from one data model to the other and changes a copy of the data accordingly. The output is a ready-to-use database dump adhering the latest data model version.

Acoda offers an Eclipse plugin developed using Spoofax/IMP [42].

2.3.3 Operation-based Tools

Operation-based tools describe the evolution by a sequence of applications of change operators, subsequently referred to as coupled operators. The use of coupled operators facilitate the reuse of recurring migration knowledge. However, adequate tool support also has to provide expressiveness to cater to complex migrations [34]. The operation-based approaches which fulfills both the reusability and the expressiveness requirements is Herrmannsdörfer's [32] tool called COPE (COuPled Evolution). COPE, however, is used to automate model

[8] http://www.acceleo.org

evolution. Nonetheless, its design properties can be of interest in designing a data model evolution variant.

There are also various Computer-Aided System Engineering (CASE) tools that support graphical data model evolution. The research tool DB-MAIN is one of such CASE tools which is specialized in database transformations. The DB-MAIN tool uses predefined schema transformations to achieve its adaptations, hence it classification as an operation-based tool.

COPE: [9] This is an operator-based coupled evolution approach for the Eclipse Modeling Framework [7]. In other words, it records coupled evolutions of meta-models and models in an explicit history model, in order to obtain a correct model transformation. As a result, the history model stores the sequence of coupled operations that have been performed during evolution. A coupled operation is defined as the meta-model adaptation and reconciling model transformations to conform the model to the new meta-model. To further reduce migration effort, COPE provides high-level operations which have built-in meaning in terms of the migration of models [36]. Furthermore, the tool supports functions to inspect, refactor and recover the history model to better understand, correct and reverse engineer the coupled evolution [32]. COPE has been transformed to Eclipse EMF Edapt [10] in 2011. The main difference is that Edapt specifies migrations directly in Java instead of through a general-purpose scripting language.

The advantages of the tool are that it allows developers to specify custom mappings when no operator is appropriate. This is done using Java. It is therefore expressive enough for complex model transformations. It uses reusable coupled transactions that encapsulate recurring transformation knowledge. The drawback is that it needs to be integrated into the editor being used to modify the model.

DB-MAIN: This CASE tool also supports the development, reengineering, migration and evolution of data-centered applications. This is done through features such as data-modeling support and transformation and generation capabilities [11].

This tool provides more than 25 schema transformations sufficiently rich to encompass most database engineering processes. These transformations are designed to automatically construct the mapping between different schemas [10].

One of this tool's features is the ability to generate a script or database which conforms to a target model given a database and the model of the database.

2.3.4 Tool Overview

Table 2.1 gives an overview of the co-evolution tooling options described above.
The DB-MAIN supports both textual and graphical model notations. However the schema transformation are intra-schema transformation, meaning they do not provide a mapping

[9] COPE web-site: http://cope.in.tum.de
[10] Edapt web-site: http://www.eclipse.org/edapt/

Approach \ Domain	Coupled Evolution Of		
	Meta-model & Model	**Model & Database**	**Database & Program Code**
Manual	EMF-based tools (e.g. Ecore2Ecore and Flock), MetaEdit+, Visual Studio VMSDK-based tools		
Matching-based	EMF-based tools	Acceleo, Acoda	
Operation-based	COPE	Acoda, DB-MAIN*	DB-MAIN

Table 2.1: Co-Evolution Tooling - Approach Overview

Approach \ Domain	Co-Evolution Of Data Model & Database	
	Textual Models	**Graphical Models**
Manual	EMF	GMF, Visual Studio VMSDK-based tools
Matching-based	Acceleo, Acoda	GMF-based tools
Operation-based	Acoda	CASE Tools

Table 2.2: Overview of current tools which support coupled data model evolution

from model to data migration. Therefore, it does not comply with the data model evolution as defined in this thesis. Because of this it has an asterisk behind it.

In table 2.2 an overview of co-evolution tools is given with a classification based on the types of model notations that they support.

2.4 Related Work

Coupled evolution plays a significant role in computer science and has been discussed in various areas. Earlier research primarily focused on constructing coupled evolution support for specific domains [69]. This section presents the recent research being done in the field of data model evolution.

2.4.1 Data Model Evolution in MDE

In March of 2012 Pérez-Castillo et al. published a process that aimed to tackle the problem of evolving database-centric software systems. In that particular article they dealt with legacy databases. The process consisted of a re-engineering process that follows model-driven development principles to reverse engineer Web services from legacy databases. These Web services manage access to legacy databases which, in turn, allows these databases to be used by newer systems in a service-oriented environment.

A support tool was created to facilitate the adoption of this process which was shown to reduce development effort and improve the return on investment by extending the lifespan of legacy databases [59].

In the same year, Vermolen et al. published their research on the automated reconstruction of complex meta-model operations, based on detected differences between two meta-models. They proposed a reconstruction algorithm capable of addressing the three major problems in the reconstructing of complex evolution steps; dependency (between operations), detection (finding steps that make up the complex step), and interference (steps that affect actions taken in another step or that are hidden by/from other steps). They im-

plemented this reconstruction algorithm in the tool Acoda which resulted in an enriched evolution trace from which the required data migration could be generated [71].

2.4.2 Database Evolution

Research over the last decade from DB-MAIN includes work done on forward propagation of changes in data models to the database it represents. For instance, changes made to a model should propagate to the database in a way that transforms it while maintaining the existing data rather than dropping the database and regenerating a fresh instance. The changes are applied using Data Definition Language (DDL) statements in order to take the database to its next version and support applications accessing the new version. In other words, it is a non-versioning strategy. There is no guarantee of backward compatibility.

DB-MAIN stores each model transformation in a history buffer. These are replayed when it is time to deploy the changes to a database instance. A model transformation is coupled with a designated relational transformation as well as a script for translating instance data in essence, a small extract-transform-load workflow. The set of available translations is specified against the conceptual model rather than the relational model, so while it is not a relational schema evolution language by definition, it has the effect of evolving relational schemas and databases by proxy [29].

MeDEA is another tool that supports database evolution by exposing relational databases as conceptual models and then allowing edits to the conceptual model to be propagated back to schema changes on the relational database. A key distinction between MeDEA and DB-MAIN is that MeDEA does not have a fixed modeling language or a fixed mapping to the database. As a result, the relationship between model and database is variable.

Given a particular object in the conceptual model, there may be multiple ways to represent that object as a schema in the database. Consequently, when one adds a new object to an existing model (or to an empty one), the developer potentially has many valid options for persistence. A key concept in MeDEA is the encapsulation of those evolution choices in rules. For each incremental model change, the developer chooses an appropriate rule that describes the characteristics of the database change [17].

2.5 Chapter Summary

A comparison between matching-based and operation-based tools reveals that the main advantage of matching-based approaches is that they might be fully automated. However, it has been shown that model migration cannot be automated in certain cases when the semantics of a modeling language need to be taken into consideration [38]. Furthermore, the comparison also shows that matching-based approaches are based on the reconstruction of the model adaptation. On the other hand, the operation-based approach avoids such reconstruction by executing model adaptations through operators, which better facilitate the coupling of model transformations to database co-evolution.

The tools that support database evolution through the propagation of changes between relational schemas and conceptual models, closely resemble the aforementioned data model evolution definition.

The information presented in this chapter was used to redesign the operation-based approach to address the goals that were set for this thesis. The approach redesign is described in the next chapter.

Chapter 3

OP-DEVO Approach: The Redesign

As mentioned in chapter 1, the objective of this design research project is to study the use of the operation-based approach to automate the co-evolution of data models. The focus of this study is on the inherent limitations of such an operation-based approach and how to mitigate them through a redesign of the approach.

This chapter describes the steps that were taken to design an operation-based data model co-evolution approach which aims to mitigate the inherent limitations of operation-based approaches. The concept behind the design is covered in Section 3.1, followed by a description of its various components in Section 3.2.

3.1 Framework of the Design

As stated in previous chapters, the design is based on the operation-based approach. This section explains the concept behind operation-based data model co-evolution and it describes the requirements involved.

3.1.1 Operation-based Data Model Co-Evolution

According to the literature survey [41] conducted prior to this research, the operation-based approach should be used to automate the data model co-evolution for two reasons. First, it can track model adaptations with greater accuracy. Secondly, it has a built-in mapping mechanism that can be extended to support the mapping of model transformations to database transformations.

Another aspect to discuss when considering an operation-based approach is the use of coupled operators. Coupled operators are central to the operation-based approach. The operators are basically a grouping of transformation statements that can transform both a model and another artifact simultaneously. These coupled operators can be divided into two categories: generic operators and custom operators. The generalized operators facilitate the reuse of recurring coupled changes across models. These simplify the data model evolution process by reducing it to the application of a series of predefined atomic transformations. The custom operators define scenario-specific model modifications. The use of these two kinds of operators results in a composeable model evolution. In other words, the evolution

from one model version to the next can be composed of manageable and modular adaptations, thus enabling flexible combinations of generalized coupled operators and custom coupled operators [35].

Of the operation-based approaches that were studied during the literature survey, the COPE tool (cf. section 2.3.3) was selected as the basis for this project. It is an established operation-based approach to co-evolve meta-models and their model instance(s). This co-evolution resembles the data model and database co-evolution concept, therefore it was decided to take advantage of the documented research concerning the construction of the COPE tool.

Basic Design Concept Based on the aforementioned, it became evident that an operation-based data model co-evolution approach (subsequently referred to as the OP-DEVO approach) could be possible by doing the following:

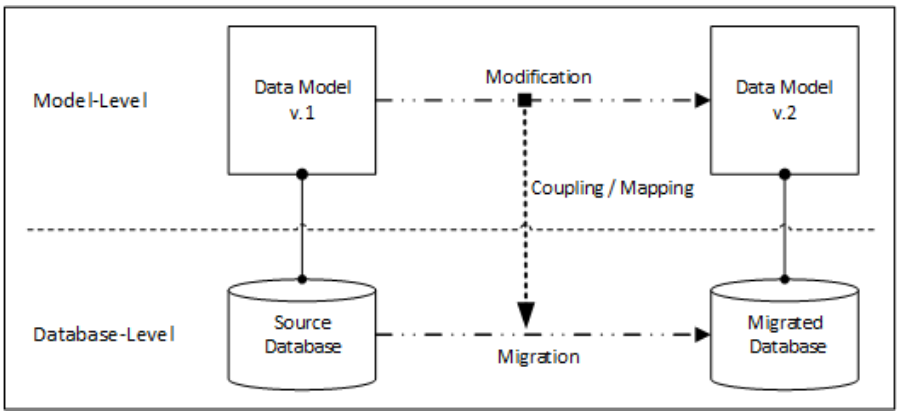

Figure 3.1: Data Model Co-Evolution Concept

1. Define coupled operators by:
   a) Collecting a list of the existing model transformations found in literature
   b) Defining a list of generalized database transformation statements; each one corresponding to a particular model transformation from step *a*
   c) Creating a mapping between each model transformation and the corresponding database transformation statements, thereby establishing an explicit link between the lists from steps *a* and *b*
2. Use these coupled operators to modify the data model
3. Modify the change recorder to not only log the transformations on the model-level but to also log the corresponding database-level transformations
4. Add a migration functionality which collects the recorded database transformations into a script. This script can then be used to migrate the existing database

In doing so, it would be possible to create an integrated and semi-automated solution for coupling data model modifications with application data migration.

3.1.2 OP-DEVO Approach Requirements

The basic requirements for the OP-DEVO (OPeration-based Data model co-EVOlution) approach were based on the work done by Herrmannsdörfer et al., who outlined four requirements that an approach should fulfill in order to profit from the automation potential [33], namely: reuse of migration knowledge, expressiveness of transformation language, modularity of operators, and recording of transformation history.

The basic requirements for the OP-DEVO approach are:

BR1 - *Parameterized Co-Evolution Operations*. The approach must use the coupled operators in its library to modify existing data models. These coupled operators must be model-independent to allow for reuse across different data models. By using parameters to detail the transformations in an operator, the operator can be independent of a particular data model [32, 31, 36, 15]. Meaning, the declaration (i.e. definition) of the operator is generic while its instantiation is model-specific.

This requirement relates to the aforementioned *reuse of migration knowledge* requirement.

BR2 - *Expressiveness, custom co-evolutions*. The operator library must be extensible. Thus, the approach ought to be flexible enough to allow custom coupled operators to be defined. Since transformations can be arbitrarily complex, the formalism used to define operators must be expressive enough to cover these arbitrary evolution scenarios.

BR3 - *Modularity*. Coupled operators ought to be specified (i.e. instantiated) independently of each other. This independency allows the different changes to be better managed, because the specification or removal of an operator will not be affected by existing operators.

BR4 - *Change Recorder*. The approach must maintain a list of the applied modifications which can then be stored for future use. This corresponds to the *history* requirement given by Herrmannsdörfer et al. They also state that such a history would be particularly useful when working with distributed artifacts.

In addition to addressing the aforementioned basic requirements, the objective of the OP-DEVO approach is to also address the inherent limitations of operation-based approaches. The limitations, of which there are five [64, 31, 44, 37], can be divided into three categories:

1. Coupled Operator Limitations:

   L1 - Operation-based approaches use a library of operators to execute the required model modification. If the existing operators cannot realize the required

model modification, then there are two outcomes. Either the approach is discarded in favor of another one that does support the needed modification, or a new operator is created to handle it. Therefore, the amount of model modification types that the approach can support is related to the amount of operators in its library. Thus, it is favorable to have a rich operator library, yet the larger the operator library the harder it is to find the correct operator for a particular task. In other words, there is a trade-off between the size of the operator library and the ease of finding the correct operator. Finding that balance between richness and navigability is a key challenge in defining libraries of coupled operators.

L2 - Most operation-based approaches use a fixed set of operators, therefore they cannot support custom transformations. From the description given in *L1* it can be deduced that a fixed set of operators means that there are a fixed amount of model modifications that such an approach can support. Therefore, it would be favorable to allow the creation of new operators instead of using a fixed operator library

L3 - This limitation is related to *L2*, in that allowing the creation of custom operators means providing the developers with a modeling language that is expressive enough for them to create arbitrary operators. These modeling languages do not capture migration patterns or other reoccurring patterns, therefore the new operators do not benefit from reusing the existing transformation knowledge

2. Database Co-Evolution Limitation:

   L4 - The coupling of database migration code to a model transformation impedes independent adaptation of either the model or the database migration code. This implies that the developer needs to take the impact of the data model changes on the database into account. Meaning, at the model-level two separate modifications can lead to an equivalent result, but at the database-level they can produce two separate results. For example, the *moving* of one attribute from one element to another is the same as *deleting* that attribute from one element and creating it in the other element. However, at the database-level the *move-operation* preserves the data represented by the attribute (i.e. the operation is data-preserving) but the *delete-and-create-operation* results in data loss. Therefore, the developer must be constantly aware of the impact of the model modifications on the database.

   Furthermore, because of this coupling the order of the model modification can affect the generated database transformation. If a modification order leads to an incorrect database transformation this would mean that the performed model modifications will have to be undone and then repeated in a different order to fix the issue. Therefore, it would be beneficial to know which modifications caused the erroneous database transformation so that only those modifications can be fixed instead of having to undo everything until the error is gone

3. Tooling Limitation:

> **L5** - In order to use an operation-based approach, the chosen modeling tool must implement the approach's operator library, the model change recorder to register the model modifications, and the migration functionality responsible for the co-evolution process. Thus, it requires a significant upfront implementation investment

The two main aspects of an operation-based approach are its operators and its mapping mechanism. The operators ultimately determine the supported co-evolution scenarios and the mapping is responsible for actually outputting the co-transformation that resulted from the model modifications. Therefore, this project will focus on the limitations associated with the definition of coupled operators (cf. L2 & L3) and the generation of database migration code (cf. L4).

3.2 The OP-DEVO Design

This section presents an overview of the OP-DEVO design starting with the design goals and followed by a dissection of the design into its various components.

3.2.1 OP-DEVO Design Goals

The aforementioned limitations, requirements and project objective were incorporated into six design goals:

- Ability to create graphical data models
- Ability to modify data models using coupled operators
- Ability to create independent coupled operators (cf. BR1 & BR3)
- Ability to extend the operator library (cf. BR2)
- Ability to record the history of data model modifications (cf. BR4)
- Ability to generate database migration code automatically based on recorded model modifications
- Ability to view and adapt the generated migration code

3.2.2 Component Interaction

There are three main components that constitute an operation-based approach: the use of coupled operators, the operator library which contains these operators, and the recorder that logs all the changes. The OP-DEVO approach contains these three components.

The interaction of these components is straightforward, in that it starts with the operator library which is a collection of all the available coupled operators. Thus, the operator

library represents the type of co-evolution scenarios that the approach can handle. The coupled operators perform the actual model transformations and generate the corresponding database transformation statements. Coupled operators are defined using a modeling language (subsequently referred to as the operator definition language). The third component, the change recorder, logs each model modification and its generated database transformation statement(s). These recorded items can be kept for future use (e.g. audits) and they can also be used to debug (i.e. identify and adapt recorded items) the approach.

Figure 3.2 illustrates the component interaction.

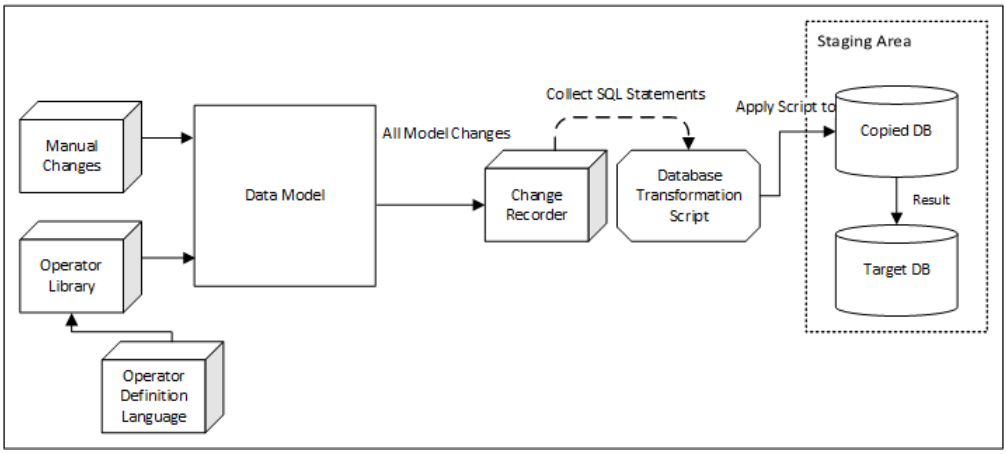

Figure 3.2: Overview Component Interaction

In the next sub-sections each component is described separately. The descriptions focus on the original shortcomings of the component and the concept behind its redesign.

3.2.3 Operator Definition Language

The type of languages that is generally used to define new operators is based on a combination of a modeling language specific to the modeling domain and a general-purpose language such as Java or C#. These modeling language combinations provide the flexibility needed to define arbitrary operators, but they do so at a cost (cf. limitation **L3** in Section 3.1.2). Therefore, the added flexibility enables the creation of any type of operator, while inhibiting the ease of its creation because it replicates the existing modeling knowledge instead of reusing it. Thus, the standard operator definition language has some room for improvement.

Operator Composition Framework The idea for defining an operator composition framework to improve the way in which operators were defined came from the realization that, the sequential application of operators can be seen as a composition of manageable, modular transactions. Such a composition can enable flexible combinations of both existing operators and custom model transformation statements [35]. In other words, the operators

are already being used as composite units of transformation. The new operator composition framework merely formalizes this concept.

The new framework presents each operator as parameterized functions that can be chained together. These functions encapsulate the various model transformation statements into a single function definition. Thus, increasing the conciseness involved in defining coupled operators. The ability to chain these functions together supports the notion of applying transformations in a particular order and more importantly it provides a knowledge-retaining framework with which to define new operators. In other words, the end result of this new framework is that new operators can be defined by encapsulating transformation statements into a single function. These functions can also define an operator that consists of multiple existing operators. Such an operator is called a composite operator. This reuse increases the precision of the new operator, because the existing operators have already been tested for correctness.

In order to define an operator composition a composition framework is needed that has semantic and syntactic specifications that are compatible with those of the operators used in the modeling environment. Furthermore, a composition theory is needed in order to reason about the composition. Such a theory enables calculation and result prediction of applying a composition to an operator [52]. The new framework will adhere to these specification in the following manner. Its syntax will include the use the dot-character (.) to indicate composition. The dot-character will always be placed between two operators. The semantics thereof is that the operator to the right of the dot-character will be executed directly after the first operator. This implies that the second operator will use the outcome of the first operator (i.e. the updated data model) as its input.

Thus, the operator composition framework facilitates the creation of new operators by allowing both the writing of transformation code and the reuse of existing operators. This should result in a more concise and precise way to define new operators, while retaining the flexibility provided by the general-purpose language that is used.

3.2.4 Coupled Operator

The coupled operators found in the literature are for meta-model and model co-evolution. Thus, a coupled operator variant is needed to perform data model co-evolution. According to the basic design concept outlined in sub-section 3.1.1, the needed coupled operator consists of three parts: the model modification operation, the database adaptation operation, and the mapping mechanism that links each model modification operation to its corresponding database adaptation operation.

Model Modification Operation Based on the work done in the field of model co-evolution, it can be stated that there is a set of model modification operations that can be considered to be complete. Furthermore, this set of possible model modifications can be divided into four categories: structural modifications, non-structural modifications, complex modifications and semantic modifications. The model modifications and the categories are based on the *practically-complete* list of model co-evolution operations as defined by Herrmannsdörfer

et al. [39]. Table A.1 in Appendix A depicts a list of model modification operations per category.

Database Adaptation Operation While searching for standardized database transformation operations, the work done by Ambler et al.[4] was found. They outline a list of database schema refactorings and transformations which they present as "best practises". A database refactoring is described as a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. In other words, a database refactoring should not add new data or change the meaning of existing data.

Each refactoring description consists of a before-and-after example, a list of reasons for applying the refactoring, a brief statement of the potential tradeoffs, the Data Definition Language (DDL) code for updating the schema, and the Data Manipulation Language (DML) code for migrating the data.

The assumption was made that the refactorings could be used to realize the needed database adaptation operation. Based on this assumption and because of the semantics preserving property of the refactorings, it was decided to use them. Thus, each model modification was paired with one or more refactoring. A list of these refactorings along with their model modification counter-part can be seen in table A.1.

Mapping Mechanism As for establishing an explicit link between the two operation types, this can either be done within the operator definition [34] or it can be done at the recorder-level by matching incoming model modifications to their database adaptation counter-parts.

The advantage of having the both the model-level and database-level transformation code in the definition of a coupled operator is that it is self-contained and it is in one place. Thus, it facilitates maintenance. The advantage of having a mapping at the recorder-level is the opportunity to also link manual model modifications to a database refactoring.

3.2.5 Operator Library

The set of operators represents the different types of co-evolution scenarios that the approach can handle. Consequently, if there is a scenario that the existing operators do not support, a new operator must be defined.

The new operator library, which replaces the old one, differs in only one aspect from the standard library in that it contains functions that either represent a single operator or a chained sequence of operators.

3.2.6 Change Recorder

The standard recorder's task is to record the tracked model adaptations so that they can then be used to produce a corresponding co-evolution strategy. The adaptations are recorded as they are applied, therefore, the produced co-evolution strategy follows the same adaptation order. Furthermore, the recorder does not provide any means to quickly locate the model

modification that produced the particular transformation and any attempt to modify database transformations will have to be by adapting the database migration script manually.

Event-based Change Recorder The objective of this component is twofold: to facilitate the search for the model transformation(s) that produced a particular set of database transformation statements and to facilitate the adaptation of the aforementioned statements without affecting anything else.

The guiding hypothesis behind the realization of this component consists of three parts. The first is that it should be possible to locate a particular event by indicating the data model entity and the modeling task that is involved. The second is that by storing model adaptations as events that consists of two separate yet connected transformations, it should be possible to adapt the produced database transformation without affecting its corresponding model transformation, while retaining the link between the two. That link needs to remain intact because it can be used to trace a database transformation back to the model adaptation that produced it. The third is that by storing both the database transformation outcome and the model-information necessary to recalculate that stored outcome, it would be possible to adapt the database transformation outcome without affecting the integrity of the event log (i.e. the outcome can be reset by recalculating it). This would make it possible to de-couple the model transformation from its corresponding database transformation.

The design of this recorder resembles the event sourcing pattern as discussed by Fowler [22]. Each model modification is logged as an event object. Such an event object contains information concerning the context of the model transformation and a timestamp. The transformation context consists of the modified model element both before and after the operator is applied. The before-the-change element value is used to locate the existing database items containing that value. The after-the-change values are used to update the database items. Furthermore, the event sourcing pattern states that the events should be are stored as a sequence, which can therefore be re-played as a sequence by simply looping through the recorded changes. This facilitates the re-application of the event on an object. The component uses this re-play mechanism to generate a database migration strategy (i.e. migration script), by collecting the set of database transformations into a single file as they are being looped through. The script provides a means to (re)view the database transformations as a whole and it provides an opportunity for manual adaption before it is executed.

Thus, the event-based recorder enables one to have more control over the recorded data, thereby enabling one to have control over the database transformations. That control is defined as the ability to identify specific recorded transformations and the ability to adapt database transformation without having it affect the existing data model.

3.3 Chapter Summary

As mentioned before, the main objective of this thesis was to find a way to mitigate two specific limitations: the operator creation limitations and the database transformations adaptation limitation. This redesign set out to mitigate these limitations by revising the two components that are directly responsible for operator creation and transformation adaptation: the

operator definition language and the event-based change recorder. These two components essentially constitute the novelty of the OP-DEVO approach.

The next chapter presents how these concepts are to be implemented by looking at the tool that was built based on the redesign. The tool is called Modler.NET.

Chapter 4

Modler.NET: The Implementation

Modler.NET is the tool that was developed to implement the OP-DEVO approach that was presented in Chapter 3. This chapter presents an overview of the implementation structure and the techniques used to realize it.

Section 4.1 describes the modeling platform on which Modler.NET was built. This is followed by an overview of the integration of the approach with the modeling platform in Section 4.2. Lastly, an outline is given of the implementation of the tool's main components and its features in Sections 4.3 and 4.4.

4.1 Modler.NET's Modeling Framework

Modler.NET was written in the C# programming language. It was built using Visual Studio's Visualization & Modeling Software Development Kit (VMSDK) [1]. VMSDK provides a platform with which custom visual editors can be defined. Such editors contain features that support the development of domain-specific models, such as point-and-click model modification, model validation and artifact generation.

4.1.1 VMSDK

VMSDK caters to graphical models and as such it features a visual interface. Furthermore, it uses a visual modeling language that is based on the conventions of the UML standard. The modeling language consist of various shapes and connectors that represent basic reusable building blocks that can be dragged-and-dropped onto its visual editor. The shapes represent domain classes which, in turn, represent the concepts in the specified domain. The connectors represent domain relationships which are a representation of relationships between domain concepts.

Domain constraints are also an import part of the platform. Constraints ensure that the created model is (semantically) valid. It is possible to define custom constraints in VMSDK using hand-written code.

[1] VS VMSDK website: `http://archive.msdn.microsoft.com/vsvmsdk`

After having created a valid model it is generally the case that some artifact whether it is source code, data, a file or even another model be generated from the created model. Platforms such as the VMSDK allows the developer to regenerate artifacts efficiently whenever the model is changed [12].

4.1.2 Managed Extensibility Framework

VMSDK offers an extension framework called Managed Extensibility Framework (MEF) [2] which facilitates the discovery and use of extensions without the need for configuration. It also supports the encapsulation of code in order to avoid hard dependencies. Furthermore, MEF allows extensions to be reused within the same application as well as across different applications. This facilitates the publishing of tool components built using the VMSDK.

4.2 Overview of Implementation Structure

VMSDK was the only established .NET-based platform (at the time of this writing) with which to easily create a custom visual editor that could meet the various design goals for the OP-DEVO approach (cf. Section 3.2.1). This is due to the fact that it supports the development of graphical data models. It also provides a declarative interface, in the form of a pop-up context-menu, with which model-transforming commands can be chosen and executed directly on selected model items. Such a context-menu can be re-purposed to contain coupled operators. In addition to that, the SDK's modeling language is based on C#, therefore, it should be flexible enough to develop parameterized operators and an operator composition framework. Furthermore, the VMSDK platform provides a mechanism through which every change to a model element can be registered. Thus, the registration mechanism can serve as the foundation for the required change recorder. Lastly, windows forms can be built to view and adapt the recorded migration information.

4.2.1 Component Data Flow

Figure 4.1 illustrates the data flow between the components and the general structure of the Modler.NET tool.

[2] MEF web-site: http://msdn.microsoft.com/en-us/library/dd460648

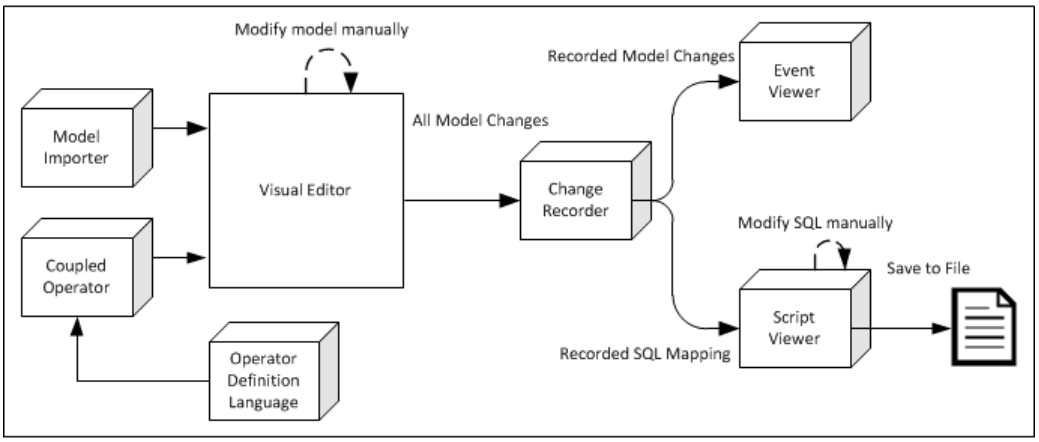

Figure 4.1: Overview Modler.NET's Data Flow

4.2.2 VMSDK Integration

This sub-section details how the aforementioned components were realized and integrated with the VMSDK platform.

Visual Editor The implementation of the visual editor was straightforward. A windows form was created that consists of: a panel (i.e. an isolated area on the form) which contains the data model elements, a column of buttons used to invoke a certain functionality and a section which shows the recorded model modifications. To register this form with the VMSDK platform as its visual editor interface, the developer is required to write six simple lines of code. The VMSDK then provides the list of available model elements, as a toolbox, that can be dragged-and-dropped onto the panel on the form.

The toolbox of available model element is generated from the meta-model that needs to be defined. This meta-model defines the syntax (i.e. the structure) of the type of data model that the visual editor will support. Based on the given syntax the VMSDK will: populate the toolbox with the correct type of model elements and generate basic model validation-rules to enforce proper model syntax.

Operator Library As mentioned before, right-clicking inside the panel area shows a context-menu. This context-menu is re-purposed to represent the tool's operator library. As the name states, the pop-up menu is context-aware. This makes it possible to implement filtering of the coupled operators that are shown in the context-menu. The filtering is based on the selected model elements or the type of model element which the mouse-pointer is hovering over when the right-click occurred.

In order to benefit from the extensibility feature of VMSDK, the operator library invocation and filtering are done in MEF. This makes it possible to add more operators to Modler.NET's library without having to register them with the VMSDK platform [53].

MEF allows each coupled operator in the library to be define as a class. These classes provide access to the selected model elements through its *SelectionContext* property. It

also provides a method with which the desired filtering can be specified. Furthermore, it provides a means to name the coupled operator and it has a method in which the coupled operator itself can be instantiated. Listing 4.1 shows the MEF class for displaying the entity rename operator when the selected model element is an entity.

```
1  public class RenameEntityCommandClass : ICommandExtension
2  {
3    // Provides access to current document and selection.
4    [Import]
5    IVsSelectionContext SelectionContext { get; set; }
6
7    // Called when the user selects this command.
8    public void Execute(IMenuCommand command)
9    {
10     Operators.Define(SelectionContext.CurrentStore).
11         RenameOperator(SelectionContext.CurrentSelection(Entity));
12   }
13
14   // Filtering based on selected model element type
15   public void QueryStatus(IMenuCommand command)
16   {
17     command.Enabled=command.Visible=(SelectionContext.
           CurrentSelection(Entity);
18   }
19
20   // Determines the text of the command in the menu.
21   public string Text
22   {
23     get { return "Rename Element"; }
24   }
25 }
```

Listing 4.1: Menu definition in MEF for the rename operator

Coupled Operator The coupled operator declarations are done in a separate class file. The model transformation statements are constructed using VMSDK's modeling language. They are created as methods that receive the specifics of the model element that needs to be modified. As for the mapping mechanism, the choice was made to use a recorder-level mapping in order to support manual model modifications. The database transformations were defined in a separate class file as well to simplify the architecture. Their implementation is similar to the model transformation side, in that they are methods containing the generic parameterized SQL code that receive the model element specifics needed to fill in the parameterized code.

Capturing Model Changes The VMSDK has change-events that are triggered when a model element is create, removed and updated. The change recorder component was implemented by creating event handlers that register with a particular change-event. In other words, the recorder implementation can be summarize as the creation of notification handlers which are fired when the VMSDK's change-events are triggered.

When a model modification triggers a change-event, that event gets raised and is propagated up to the event handler which in turn passes the event information to a method that will record the information.

Listing 4.2 shows how the event handlers for model entities were registered. Two of these registered handler methods are shown in listing 4.3. The call to the method *handleChangeNotification()* takes two parameters: the state and the corresponding mapping function. Hence, this is where the model transformation gets paired with its database transformation statements.

```
1  // Handlers are added to the various change-events of the EMD:
2  EventManagerDirectory emd = store.EventManagerDirectory;
3
4  // Store events are registered per domain class, not per instance.
5  // After a listener is registered with a class, it is called for
       every change to any instance of the class
6  DomainClassInfo entityInfo = store.DomainDataDirectory.
       FindDomainClass(typeof(Entity));
7
8  emd.ElementAdded.Add(entityInfo, new Event<ElementAddedEventArgs>(
       logger.elementAdded));
9
10 emd.ElementDeleted.Add(entityInfo, new Event<
       ElementDeletedEventArgs>(logger.elementRemoved));
11
12 // Set one handler for all properties of a class
13 emd.ElementPropertyChanged.Add(entityInfo, new Event<
       ElementChangedEventArgs>(logger.elementUpdated));
```

Listing 4.2: Register handler for event triggers

```
1  void elementAdded(object sender,ElementAddedEventArgs e)
2  {
3    EventContext eventContext=fillEventContext((e.ModelElement).Name
         , ElementCreatedString, e);
4    // log the state and SQL mapping for this particular event
5    handleChangeNotification(new AddedState(eventContext), new
         AdditionMapping(eventContext));;
6  }
7
8  void elementRemoved(object sender,ElementDeletedEventArgs e)
9  {
10   EventContext eventContext=fillEventContext((e.ModelElement).
         CurrentName, ElementRemovedString,e);
11   // log the state and SQL mapping for this particular event
12   handleChangeNotification(new RemovedState(eventContext), new
         RemovalMapping(eventContext));
13 }
14 ...
```

Listing 4.3: Event Handlers

4.3 Component Implemention Details

This section details how the event-based change recorder, the operation composition framework and the operator library were implemented.

4.3.1 Change Recorder

To record the coupled evolution, a recorder is seamlessly integrated with the visual editor used to perform the data model modifications. The recorder ensures that all changes to the model's state are stored as a sequence of events and that each event can be queried separately. These two properties make it possible to reconstruct past states using the event log and it also enables the automation of state adjustments in order to cope with retroactive changes.

The recorder captures every state change in an event object and stores them in the same sequence as they were applied. The key is to guarantee that all changes are initiated by the event objects.

The main implementation challenge for this component was determining the event object's structure (i.e. how the events would be recorded). There needed to be a tradeoff between the ease with which the information is stored and the ease of its retrieval.

The decision was made to construct one entity-event object for each modified model entity which consists of an entity, an attribute and an inter-entity relationship (subsequently referred to as a link) segment. Each of these three segments represents a sequence of event-context objects. An event-context object is simply a set of properties such as the timestamp, the affected model element type, the model element data values, and a hook into the mapping mechanism for that event. Listing 4.4 illustrates the properties of an event-context object.

```
1  // Context of a model event
2  public struct EventContext
3  {
4    public string actionText;
5    public string entityName;
6    public DateTime timestamp;
7    public bool doSkipThisEvent { get; set; }
8    public MappingBase sqlEventMapping { get; set; }
9    public ContextElementType elementType;
10   public ElementEventArgs elementEventArgs;
11 }
```

Listing 4.4: Context Information for each Event Object's Segment

In addition to the aforementioned segments, an entity-event object also has a 'MainEntityName' property which allows for easy retrieval of the name of the parent entity of the affected model element(s). Furthermore, the event object also contains a SQL script generator. This generator is a method that loops through the recorded events and collects all of the mapped database adaptation code. The collected code is used by other components which display it.

This event structure allows each change-event to be grouped into the entity that it pertains to, thereby making the retrieval of all events for a particular model entity trivial. The retrieval of events affecting a particular type of model element is also made trivial since each change-event is separated into one of three segments depending on the affected element type. The storing of the change-events, however, requires a bit of overhead in that the recorder needs to identify or create the main entity-event object into which the change-event is to be stored. Once the entity-event is established, the change-event is converted into an event-context object and added to the collection of its respective segment.

The code for the entity-event object is shown in listing 4.5.

```
1  public class EntityEventInfo
2  {
3    // Events that pertain to an element
4    public IList<EventContext> entityEvent;
5    public IList<EventContext> attributeEvent;
6    public IList<EventContext> linkEvent;
7
8    public ElementEventInfo()
9    {
10     this.entityEvent = new List<EventContext>();
11     this.attributeEvent = new List<EventContext>();
12     this.linkEvent = new List<EventContext>();
13   }
14
15   // Name property of the main entity
16   public string MainEntityName { get; set; }
17
18   // Collection of the SQL mappings
19   public string collectSQLStatements()
20   {
21     StringBuilder strbResult = new StringBuilder();
22
23     var allSQLMappings = (from eContext in entityEvent where
           eContext.doSkipThisEvent == false select eContext.
           sqlEventMapping).
24       Union(from eContext in attributeEvent where eContext.
             doSkipThisEvent == false select eContext.sqlEventMapping
             ).
25       Union(from eContext in linkEvent where eContext.
             doSkipThisEvent == false select eContext.sqlEventMapping
             );
26
27     foreach (MappingBase sqlMapping in allSQLMappings)
28       strbResult.Append(sqlMapping.GetSQLCodeMapping);
29
30     return strbResult.ToString();
31   }
32 }
```

Listing 4.5: Structure of an Entity-Event Object

4.3.2 Coupled Operator

This sub-section illustrates the implementation of parameterized coupled operators, by focusing on the declaration of the entity/attribute renaming operator.

Rename Operator renames a model element (i.e. entity) or an attribute. Listing 4.6 shows the declaration code for this operator. The rename coupled operator is defined using .NET's generics. The use of generics allows it to be used to rename either an entity or an attribute based on how it is instantiated. The actual element renaming is handled by separate methods to facilitate maintenance.

```
1  public Operator RenameOperator<T>(IList<T> elementCollection)
2  {
3    // Constraint Validation Method
4    checkRenameOperator(elementCollection);
5
6    if (elementCollection != null)
7    {
8      if (elementCollection contains element-definition objects)
9        renameEntity(elementCollection);
10     else if (elementCollection contains attribute-definition
           objects)
11       renameAttribute(elementCollection);
12   }
13   return this;
14 }
```

Listing 4.6: Rename Operator - Pseudo Declaration Code

The Renaming Method The input required for the actual renaming of an entity is the new entity name. The new name is retrieved by prompting the user (through a input dialogbox) to type it into a textbox. Listing 4.7 illustrates the implementation of this renaming process. Furthermore, it also shows that the method is capable of handling the renaming of multiple model entities.

The main implementation challenge associated with this component was to figure out which model information to pass to the operator functions. In other words, the choice of parameters for each coupled operator function was tricky. VMSDK provides change-objects that contain the data values of the changed model element. The properties of these objects are filled in differently dependent on the type of model change that occurred. For example, the removal of a newly created element using the keyboard's delete button or using the undo functionality effectively achieves the same outcome (i.e. the removal of the created element). However, the property values of the change-object that indicates which element was removed are different when using these two removal methods. When the keyboard is used the change-object's properties are filled in, i.e. the name and the Globally Unique Identifier (GUID) of the removed model element. When using the undo function, however, the change-object has no property values. The work-around for this issue was to define multiple function signatures for each operator, where each function contained different parameters.

Each of these extra function signatures converts its parameter(s) as needed and then they invoked the same main function definition, passing along the corrected parameters.

```
1  private void renameEntity<T>(IList<T> entityList)
2  {
3    // Transaction is required if you want to update elements.
4    using (Transaction t = BeginTransaction("Rename Entity"))
5    {
6      string strElementName = string.Empty;
7      foreach (T entity in entityList)
8      {
9        Entity element = (entity as EntityShape).ModelElement as
             Entity;
10       strElementName = Interaction.InputBox("Enter a new element
             name:", "Rename Element - Inputbox", "ElementName");
11       if (strElementName != string.Empty)
12       {
13         element.Name = strElementName;
14         strElementName = string.Empty; // reset for next iteration
15       }
16     }
17     t.Commit();
18   }
19 }
```

Listing 4.7: Entity Renaming - Declaration Code

The code for the attribute renaming method is analogous to the aforementioned entity renaming code, therefore, it is not explicitly mentioned.

4.3.3 Operator Composition Framework

The operator composition framework was defined as the appending of operators to the base operator object using the dot-character (.). The base operator object ensures that the proper states are maintained and used throughout the composition. This is done by using a single model-store object which is the in-memory store that VMSDK uses to save the current state of models.

An operator composition is instantiated with the current model-store. After the model-store has been set, operators can be appended to the base operator. Listing 4.8 gives two examples of this operator composition instantiation followed by the adding of composite operators.

```
1 Operators.Define(CurrentStore).MoveOperator(attributeList,
      targetEntity);
2 // or
3 Operators.Define(CurrentStore).
4   CreateOperator(IList eInfoList, out Guid newElemID).
5   MoveOperator(attributeList, newElemID);
```

Listing 4.8: Operator Composition

This composition was achieved through two aspects. The first was defining a partial class 'Operators' that contains the definition for the method 'Define' which ensures that the appropriate model-store is used throughout the composition. Furthermore, it contains a partial class *'Operator'* in which all the coupled operators will be defined. Since the *'Operator'* class is defined as a partial class, it is possible to split the class across separate code-files. In this case, each coupled operator was defined in a separate code-file. This design choice facilitated the maintainability of the coupled operator code and made it easier to extend the number of operators.

The second aspect was defining the return-type of each coupled operator function as type *'Operator'*. This meant that C#'s internal use of the dot operator for accessing a type's members could be used. In other words, the dot operator is used as it was intended, namely: to access specific methods within a class. In this case the class was the *'Operator'* class. Since each coupled operator was defined as a function, the internal dot operator behavior allowed each of these functions to be chained together.

4.3.4 Mapping Model Events to SQL Statements

There are two kinds of model changes that are registered as events: the first is the application of a coupled operator and the second is the updating of property-values (i.e. setting cascading mechanism or updating an attribute's data type) through the tool's user-interface.

Each event has a mapping attached to it which maps the model transformation to a database transformation. Both the model transformation and its mapping result are stored by the recorder. The benefit of this construction is that the migration script can be generated at any time by simply running through the recorded events and collecting the mapping results.

Current CASE tools can automatically generate DDLcode (i.e. code dedicated to the creation of data structures), however, they generate incomplete DDL code that must be modified to be operational. Additionally, data conversion are left for the developer to implement [40].

Modler.NET not only produces DDL and DML code to provide a more complete database transformation, but it also handles data conversions. The following sub-sections address these claims.

Coupled Operator Mapping

Listing 4.9 shows the database adaptation implementation for the entity creation operator.

```
1  private void addTableSQLCode()
2  {
3    // the element information is retrieved from the stored event
         information
4    Entity entity = eventContext.elementEventArgs.ModelElement as
         Entity;
5
6    strSQLMapping.AppendFormat(@"CREATE TABLE [{0}].{1} (" +
         Environment.NewLine, Data.DatabaseSettings.SHEMA_NAME,
         entity.Name);
7    strSQLMapping.AppendFormat("    {0} " + MappingTypeConvertion.
         translateDataTypeIntoSQLString(SqlDbType.NVarChar),
         strTempColumnName);
8
9    foreach (EntityAttribute attribute in entity.EntityAttributes)
10   {
11     strSQLMapping.Append(@"," + Environment.NewLine);
12     strSQLMapping.AppendFormat("    {0} {1}", attribute.Name,
           MappingTypeConvertion.translateDataTypeIntoSQLString(
           attribute.Type));
13   }
14
15   strSQLMapping.AppendLine(Environment.NewLine + ");" +
         Environment.NewLine + "GO" + Environment.NewLine);
16 }
```

Listing 4.9: Implementation of the Database Adaptation - Entity Creation

Property-value Update Mapping

As for setting property values, the implementation is similar to the mapping for an operator with the exception of updating attribute data types.

This attribute data type change translates to a changing of the corresponding column's data type, because an attribute represents a column in a database table. The changing of a column's data type is more involved than just setting a value. It involves what are called implicit and explicit data value conversions. In other words, the data values in a column must conform to a column's data type property, or else the database is said to be in an invalid state. Therefore, these data values need to undergo a conversion.

An implicit conversion means that the database itself can safely (i.e. without loss of information) convert the existing data values so that they can conform to the specified data type. Explicit conversions, however, require extra input in the form of a conversion function. These functions should provide the appropriate conversion of the data values. Appendix D contains an overview of all the supported data type conversions per conversion type.

Modler.NET handles explicit conversions by copying and simultaneously converting the existing data value into a temporary column that uses the new data type. If the copying is successful, then the original column is dropped and the temporary column is renamed to replace the original column. Hence, the detected attribute remove and rename patterns. If the explicit conversion attempt fails, the developer receives an error which will require user

input to correct. It must be noted that since the original column was copied, no data is lost as a consequence of the failed conversion.

Listing D.1, which is located in Appendix D, contains a partial representation of the function responsible for providing the data type conversion mapping.

4.4 Modler.NET's Features

Figure 4.2 depicts Modler.NET's main user interface. The user interface can be divided into four compartments. The compartment on the left-side has two sections. The first labeled "Toolbox" contains the basic building blocks for designing a model. These building blocks can be dragged-and-dropped onto the visual editor (also referred to as the drawing canvas) which is the center compartment. The second is the properties section which displays the properties and the values of the selected model elements. The compartment on the right-side contains two sections as well. The top-right section has buttons for the tool's various functions. The bottom-right section houses a check-box for pausing the change recorder, and a tree-structured representation of the logged events.

This section contains a description of these features as well as the operator library UI.

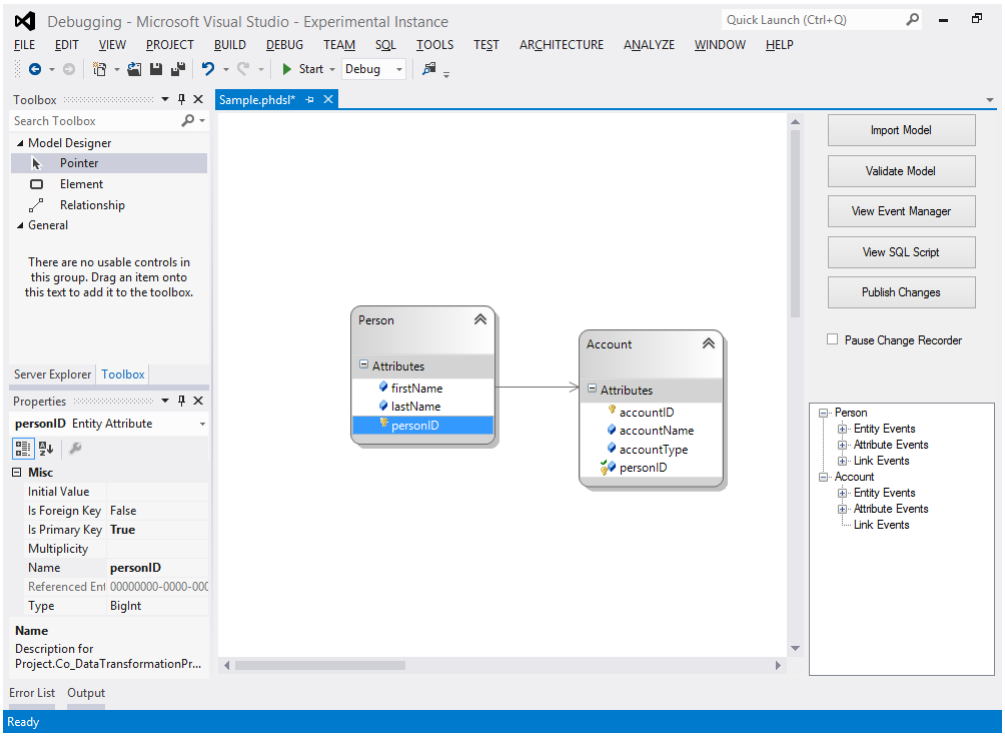

Figure 4.2: Modler.NET's User Interface

4.4.1 Visual Editor, Model Toolbox & Event History

The model toolbox is a standard feature. The toolbox contains element-tools and connection-tools. The former represents a graphical notation of each type of model element that is supported by the specified domain. The latter represents the types of relationships that the domain specifies. These are provided at run-time.

The visual editor, as mention in Section 4.2.2, is a form that is used as a visual editor. This form is treated as a drawing canvas at run-time. VMSDK also provides drag-and-drop functionality between the toolbox and the visual editor. As part of the editor, the tool has a properties-section which simply displays the properties of the selected model element.

The event history section in the lower right-side of the tool's UI was built to displays the modifications that have been recorded. The recordings are grouped by entity and by the type of model element that was involved in the modification.

4.4.2 Operator Library

The coupled operators are applied to the model through the visual editor's context-menu feature, which represents the operator library. Figure 4.3 shows the operator library menu that appears when an element in the editor window is clicked using the right-mouse button. In that figure the mouse was hovering over a model attribute, therefore the operator library is filtered to contain operators applicable to model attributes. This use of context-awareness reduces the selection complexity issue by filtering the whole operator library down to only the relevant operators and thereby making it easier to select the required operator.

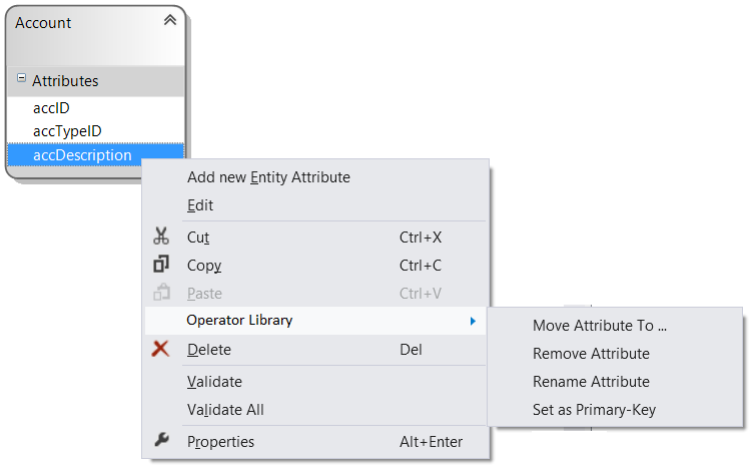

Figure 4.3: Context-aware Operator Library UI

4.4.3 Model Importing

Due to the *data model creation* design goal, a model importation function was built to automate the process of creating a data model. The model importing function (as well

as all the others) was implemented using the Strategy Design Pattern[3] to ensure that the functionality is extensible. This meant that the main code file consisted of an abstract class with abstract methods which needed to be overwritten by a concrete class with specific importing-logic.

Listing 4.10 contains the code for the ModelBuilder abstract class. It consists of three methods: the first is used to establish a connection with the source of the data model information, the second is used to create the entities and attributes, and the third creates the references between the entities.

```
1  public abstract class ModelBuilder
2  {
3    // Model root is used to access the diagram's element store
4    private static ModelRoot modelRoot;
5    // Flag to determine if the tool is in import-mode
6    public static bool isImporting = false;
7
8    public virtual ModelRoot objModelRoot
9    {
10     get { return modelRoot; }
11     set { modelRoot = value; }
12   }
13
14   // Method for connecting to the model data source
15   public abstract void BuildDiagram();
16   // Method for creating entities
17   public abstract System.Guid createEntity<T>(T entity, bool
         hasForeignKey, string fkAttributeName, System.Guid
         referencedEntityId);
18   // Method for creating references between 2 entities
19   public abstract void createLink(Entity elementA, Entity elementB
         , string attributeInA, string attributeInB);
20 }
```

Listing 4.10: ModelBuilder Abstract Class

4.4.4 Model Validation

To ensure that the generated migration script was based on a valid model, there needs to be a validation function. VMSDK inherently enforces syntax through syntactical constraints, but not semantic ones. For example, it does not enforce unique model element names. Because model elements represent database tables and database tables are not allowed to have similar names, custom validation rules were needed.

Model Consistency The consistency of a model relates to two things: the syntactic and the semantic validity. The syntax consistency of a model entails ensuring that the evolved model fulfills the constraints defined by its meta-model. The semantic consistency of a data model entails ensuring that the model changes are allowed by the Database Management

[3]Reference web-site: http://www.oodesign.com/strategy-pattern.html

System (DBMS) that manages the data represented by the data model and it also entails that the model changes preserve referential integrity.

Modler.NET enforces syntactic data model consistency through the provided validation rules. The only custom rules that were implemented are that entities, and attributes within the same entity, must has unique names and that an entity must have at least one attribute.

As for semantic consistency two rules were implemented. The first is that to establish an association between entity A and entity B, the source entity (in this case entity A) must have a primary-key defined. Modler.NET will then automatically generate the needed foreign-key constraint or ask the user to provide one. The second is that changing an attribute's data type is only allowed if the DBMS allows such a conversion.

Data Model Conformance The conformance between the data model and the data that is represents entails ensuring that the data model correctly represents the schema of its corresponding database.

Modler.NET does not enforce the conformance constraint. The work-flow was to import the data model based on meta-data from the database, therefore, it was assumed that the imported aspects (the data model and database) were in conformance. Consequently, there is no conformance preserving constraint implementation to ensure successful co-evolution completion. This was done through observation. On the other hand, not having this constraint enforcement allows for free model adaptation, meaning applying model modifications without have the tool prompting that the database no longer conforms to the data model.

4.4.5 Event Viewer

Modler.NET has a EventManager utility (cf. figure 4.4) which is built on top of the event-based recorder. This utility provides management controls of the recorded events and the generated database transformation code.

The EventManager displays a list of all modified entities. Selecting one of these entities populates the 'Event Details' section with the model modifications that were recorded for that particular entity. Besides displaying the event details it is also possible to set a 'skip-this-event' flag per event. This flag signals the script generator function to not collect the SQL statements that pertain to the event. Furthermore, by selecting a particular event detail the utility shows the SQL statements that corresponds to that event. It also allows these statements to be edited and saved.

4.4.6 Script Viewer

The Script viewer feature (cf. figure 4.5) is a windows-form which displays the generated SQL statements in an editable manner. It also has a 'Save To File' function that creates a script file containing the SQL statements.

Each time this form is opened it retrieves the generated script code from the change recorder to ensure that it is up-to-date.

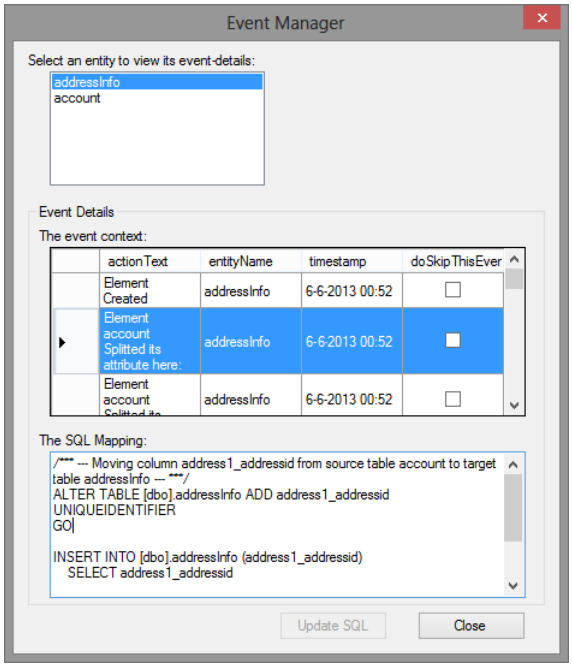

Figure 4.4: EventManager UI

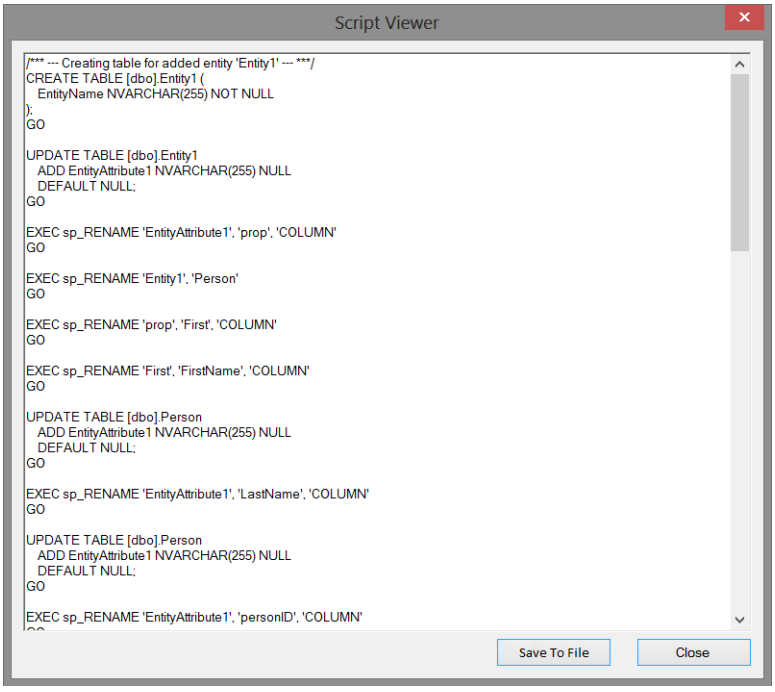

Figure 4.5: SQL Script Viewer UI

4.4.7 Publishing the Changes

The publishing of the model adaptation entails the propagation and persisting of the data model changes to the database layer. This is done in two phases. First, the database transformations are collected into a script. In the second phase the script is executed on the copy of the existing database (which is referred to the staging database).

Thus, instead of using in-place updating, Modler.NET works on a copy of the original database. The advantage of this approach is that it is conceptually simpler and undoing erroneous evolution is straightforward, i.e. delete the copy and make a new one. The drawback is that it is inherently slow when dealing with large amounts of data.

4.4.8 Pausing the Change Recorder

Modler.NET has a pause feature which when activated stops the change recorder from recording any modification. The feature is activated or deactivated by checking or unchecking the check-box above the 'event history' section. By activating this feature a flag is set that signals the change recorder to not process the change events it receives.

It is particularly useful for making small corrections in imported data models to ensure that the model conforms to the database it is representing, before starting the co-evolution process.

4.4.9 Excluded Features

In order to maintain the project schedule as much as was possible certain features were excluded from the tool. This feature freeze consisted of: the automated database migration, and the implementation of a data modeling notation standard.

Automated Database Migration The second phase of publishing the recorded model changes is the automated migration of the database. The intent was to implement this automated migration in two phases. First, all the modified entities were to be collected in order to determine which tables need to be copied. In the second phase, the staging area is created by copying the relevant tables into a temporary database (subsequently referred to as the staging database). The last phase comprises the execution of the generated script on the staging database.

Due to time constraints both phases of the automated migration were excluded. The database copying was to be done manually as well as the execution of the script on the database.

Since this feature freeze does not affect the main components of the redesign project, its exclusion was considered to be acceptable.

Modeling Notation Standard In defining the syntax for the type of data models that Modler.NET should support, the decision was made to use a modeling notation standard. Their are three main data modeling notation standards: Entity-Relationship (ER), Information Engineering (IE), and Extended IDEF (IDEF1X) [60, 28]. The ER method produces

models with clutter because of its use of separate symbols for each attribute and each relationship. Hence, it was not chosen. The IE does not have the clutter issue, however, it lacks notations for unique identifiers. IDEF1X is a hybrid language combining conceptual notions (e.g. entity, attribute, relationship) with relational database constructs (e.g. foreign keys). It is seemed to be ideal yet it is possible for the making of one adjustment to the model to result in the changing of multiple symbols.

Ultimately the decision was made to implement a simplified UML Class Diagram type notation. It consisted of simple rounded blocks for entities and black lines for the association relationship. It only supports the binary association relationship-type. To identify unique attributes a key-icon is placed to the left the attribute's name. Furthermore, the notation does not contain notations for cardinality or inheritance. Thus, it does not conform to the UML Class Diagram or the UML For Data Modeling syntax presented in work done by Ponniah [60].

The simplified data modeling notation does not hinder the execution of the model transformations listed in Section 3.2.4, therefore its exclusion was considered to be acceptable.

4.5 Chapter Summary

The implementation of the OP-DEVO approach is based on Visual Studio's VMSDK platform. VMSDK provided the integration possibilities needed to fulfill the design goals mentioned in Chapter 3. This chapter presented the methodology that was used to implement the OP-DEVO approach. The end result of this implementation process is the Modler.NET tool.

This chapter also described the implementation choices and the implementation problems encountered during the construction of Modler.NET's various components. The focus of these descriptions was on the change recorder, the coupled operator, and the operator composition framework. These components are at the center of this redesign project.

The next chapter will use Modler.NET to assess the feasibility of the OP-DEVO approach as well as to verify its claims of improved extensibility and adaptability.

Chapter 5

Evaluation

As described in chapter 1, the goal of this research project is to redesign the operation-based approach so that it could provide better support for the coupled evolution of graphical data models and their data. This improved support was to be obtained by addressing the approach's inherent limitations related to its expressiveness and its adaptability.

In order to evaluate the proposed operation-based approach a software tool, Modler.NET, was developed that implemented it. This tool was used to collect empirical data which was then analyzed and interpreted to deduce the extent to which the proposed approach achieved the research goal.

This chapter presents the evaluation of the feasibility, expressivity and adaptability of the proposed approach redesign. This evaluation was based on three exploratory case studies whose process and set up are outlined in Section 5.1. The results obtained from the conducted experiments are reported in Section 5.2. Sections 5.3 and 5.4 concludes this chapter with a validity assessment and a summary of the evaluation results.

5.1 Overview of the Evaluation Methodology

The overall objective of the evaluation was to assess three aspects. The first was the feasibility of using an operation-based approach to automate the co-evolution of graphical data models and their data. The second was the assessment of the operator composition framework's contribution to the approach's expressiveness. The third aspect that needed to be assessed was the event-based change recorder's contribution to the approach's adaptability.

The evaluation consisted of three case studies; one for each of the aforementioned evaluation aspects. Table 5.1 shows how these cases also relate to the limitations which this research project sets out to mitigate.

A notable aspect of table 5.1 is that there are no cases that assess limitation L1. This is due to the fact that it falls outside of the feasibility, expressiveness and adaptability scope. It is a usability concern that was addressed in Section 4.4.2 using reasoning. Therefore, it was excluded from this evaluation.

Approach Limitation	Case I	Case II	Case III
L1 - Operator Selection Complexity	-	-	-
L2 - Lack of Custom Transformation Support	x	x	
L3 - Lack of Knowledge Reuse		x	
L4 - Self-imposed Model Transformation Order Restriction			x
L5 - Costly upfront Implementation Investment	x		

Table 5.1: Overview - Case Study and Approach Limitation Matchup

Used Resources The graphical data models needed to conduct the experiments were generated from meta-data found in a Customer Relationship Management (CRM) application called Microsoft Dynamics CRM 2011. The CRM installation was filled with sample data which contains multiple records for a few entities [1]. Four of these entities were chosen as the basis for the generated data models, namely: the 'account', 'contact', 'lead' and 'opportunity' entities. Of the entities that were populated with data, these four entities were chosen because they represent some of the core objects in CRM applications.

The four generated data models were constructed by taking one of the four entities and importing all entities directly connected to it. Table 5.2 gives a brief overview of the properties of the generated data models. The actual data models are detailed in appendix B.

The reasons for choosing this type of data model were:

- Data model and database co-evolution is an actual recurring scenario in the development and maintenance of CRM applications. This evaluation will therefore aid in demonstrating the applicability of the proposed redesign
- Because CRM applications represent a real-life instance of the research problem, the proposed approach will be evaluated using realistic data
- Generating the models based on imported data, means avoiding having to create the needed models by hand. In other words, it saves time

The following are the additional software applications used to execute the case studies: Visual Studio 2012 (Ultimate Edition), Modler.NET (v1.0), Microsoft SQL Server 2012 (Business Intelligence Edition) and Microsoft SQL Server Management Studio (v11.0.3128.0).

The next sections outline the case study set-ups.

Model Name	Data Source	Nr. of Entities	Nr. of Attributes	Nr. of Links
IM_1	Account Related Meta-data	45	2919	46
IM_2	Contact Related Meta-data	44	2862	44
IM_3	Lead Related Meta-data	36	2332	36
IM_4	Opportunity Related Meta-data	40	2704	40

Table 5.2: Overview - Initial Data Models

[1] http://blog.xrm.com/index.php/2011/06/sample-data-in-microsoft-dynamics-crm-2011/

5.1.1 Case I - Feasibility

The objective of Case I was to determine which types of model modifications the operation-based approach can support and also to find out if it can produce the corresponding database transformations. This objective was reformulated into the following questions:

1. *Model Transformations* - How does the type of model modification affect the tool's ability to produce the appropriate target model?

2. *Database Transformations* - How does the type of model modification affect the tool's ability to produce the appropriate database transformation?

The Required Input Two inputs are needed to execute the case study. The first is the generated data models. Each of them are used to define pairs of starting models, termed initial models, and corresponding target models. These target models are an evolved version of the initial model that they are paired with. Furthermore, since each model represents a database, they all have corresponding initial and target databases as well.

The second input is the set of model changes that represents the four different types of possible model modifications (cf. Section 3.2.4). The four types of model modification are: structural modifications, non-structural modifications, complex modifications and semantic modifications. This input selection is based on a list of operators that was researched and concluded to be practically-complete [39]. Thus, the input is an unbiased representative of the different types of possible model modifications.

The choice was made to have one target model represent one type of modification. Meaning that each target model TM_1 was defined in terms of one category of model changes needed to transform its initial model IM_1 into TM_1. Therefore, it can be said that each modification type is represented by its own target model. This categorization of the target models allows for the granularity needed to answer the questions posed above.

Furthermore, data triangulation was used by varying the chosen initial models and target models. This variation was done by having two sets of input data. The first consisted of a fixed initial model (IM_1) and the type of target model paired with it was varied. The second contained a fixed target model type while the initial model was varied.

The Execution Plan The preparation phase consists of loading an initial model into the tool and using the tool to transform it into each of the target models that it is paired with.

After completing the preparation, the produced target models are compared to their predefined target model counter-parts to determine their equivalence based on:

- Having the correct entities (the name and the amount of entities)

- Having the correct attributes (the name, the data type and the amount of attributes)

- Having the correct inter-entity relationships

Since the use of an automated model comparison tool would have required the implementation and fine-tuning of a model exporting function, it would be quicker to do the comparison manually. Thus, each model modification was to be checked by hand.
The type of model modification that the tool can support is determined based on whether or not the produced model equals the target model specific to that modification type.

The final part is to use the tool to generate separate database transformation scripts based on each of the model transformations that were done and to execute each script separately on the initial database. The resulting database from each script execution is then compared to the corresponding target database using the 'Visual Studio 2012 - Schema Compare' tool [2]. This utility generates an overview of the schematic difference (per table) between the two databases.

The equivalence of the produced database to the target database is based on a few criteria, namely:

- Having the correct table names
- Having the correct columns (correctness is defined as having the same data types)
- Having the correct relationships constraints
- Having the correct migrated data values

Similar to the model comparison, the extent of the equivalence of the two databases is used to determine the type of database transformations that the tool can handle.

5.1.2 Case II - Expressiveness

With this case the objective was to determine if operator composition could increase the overall expressiveness of an operation-based approach. To this end, the following questions were posed:

1. Are there recurring transformation patterns that can be used to define an operator as a sequence of patterns?
2. Can these patterns be used as basic operators?
3. Which operators could be defined in terms of a sequence of basic operators?
4. How can the use of operator composition increase expressivity?

The Required Input The only input required for this task is the list of coupled operators used in Case I. Every item in the list of model modifications (cf. table A.1) is an operator. As mentioned, the list of coupled operators is a practically-complete list and is therefore a viable representative of the standard operators used to perform co-evolution tasks.

[2] http://msdn.microsoft.com/en-us/library/aa833202(v=vs.100).aspx

The Execution Plan The first part of the assessment is to determine if there are transformation patterns that are used to define operators. To do this, the 'Visual Studio 2012 - Code Clone Analyser' [3] utility is used to detect recurring transformation patterns in the code used to implement the aforementioned list of operators. The standard detection criteria used by the tool are that it finds "near miss" clones where it tolerates identifier renames, insert and delete statements and rearranged statement ordering. As of the time of this writing, the amount of these changes that it can tolerate was undisclosed. Furthermore, only statements in methods and property definitions are compared. That is to say, type declarations are not compared. Lastly, statement fragments with more than 40% token changes will not be detected.

The second part is to divide these detected patterns into two groups. The basic operator group will consist of operators that do not use multiple patterns to achieve their transformation. The second group consists of the composable operators. These are operators that display a use of multiple patterns.

The third part sets out to determine if Modler.NET's composition framework can produce a composite operator that can generate transformations that are equivalent to its composable operator counter-part while enabling a more expressive way to define operators. The equivalence comparison is based on the transformation equivalence of the operators in question. As for the composition framework's contribution to an increased expressiveness, this is determined by tackling three characteristics of expressiveness: conciseness (the amount of lines-of-code used to achieve an end result), precision (a measure of how straightforward the written code is, i.e. the amount of boilerplate code involved in defining a co-evolution task) and completeness (a measure of how many of the possible co-evolution tasks can be expressed). The conciseness characteristic is assess using the Lines-Of-Code (LOC) metric to compare each operator implementation method. As for the assessment of the precision, this is done through observational comparison of the implementation code. Finally, the completeness characteristic is argued using reasoning.

5.1.3 Case III - Adaptability

The objective of this case was to determine if the redesigned change recorder can increase the adaptability of the database migration process. Adaptability in this context is defined in terms of fixability, meaning the ability to fix the generated database transformation statements without affecting anything else [20]. In order to determine this, two sub-goals were defined. The first was to show that there are model transformations where it is evident that the tight coupling of model modifications to database transformations results in an erroneous database migration. The second objective was to show that the event-based change recorder enables retroactive correction of such erroneous database transformations.

The following questions were used to guide this case study:

1. How does the event-based recorder facilitate locating the event(s) that produced the incorrect SQL statement?

[3] Code Clone Analyser web page: http://msdn.microsoft.com/en-us/library/hh205279.aspx

2. How does the event-based recorder facilitate the correction of the incorrect SQL statements?

The Required Input Two inputs are needed for this case. The first is the set of model transformations that contain modifications that fit the criteria of causing database transformation errors because of the coupling of the two types of transformations. The second input are the event histories that pertain to the aforementioned model transformations.

The Execution Plan The first part of the assessment is to collect the model transformations that fit the criteria and load each one, separately, into Modler.NET.

Secondly, using the tool's EventManager feature the event log is accessed to pin-point the event(s) that produced the incorrect statements by searching for events based on the entities, attributes and links related to the transformation error.

Finally, once the event or events are found, the related SQL statements are corrected after which the tool's script generator is used to generate an updated database transformation. The success of the adaptation is determined by comparing the produced database - which is obtained by executing the correct script on the initial database - and its predefined target database.

The Visual Studio "Schema Compare" utility is used to compare these databases. The criteria for determining the equivalence are:

- Having the correct table names
- Having the correct columns (correctness is defined as having the same data types)
- Having the correct relationships constraints
- Having the correct migrated data values

5.2 The Evaluation Results

This section presents the results from each case along with observations that were made during their execution.

5.2.1 Case I - Feasibility Results

Table 5.3 gives an overview of the input data sets that were used to execute this case study. The outcome of the equivalence tests of the produced model transformations and database transformations are summarized in tables 5.4 - 5.7. This sub-section discusses each of them individually.

Table 5.4 shows that the produced model and database transformations for the *structural modifications* type met the given criteria.

The results for the *non-structural modifications* (cf. table 5.5), are similar to those from the previous modification category with the exception of two differences between the

produced database and the target database. Figure 5.1 depicts these difference as presented by the Schema Compare utility.

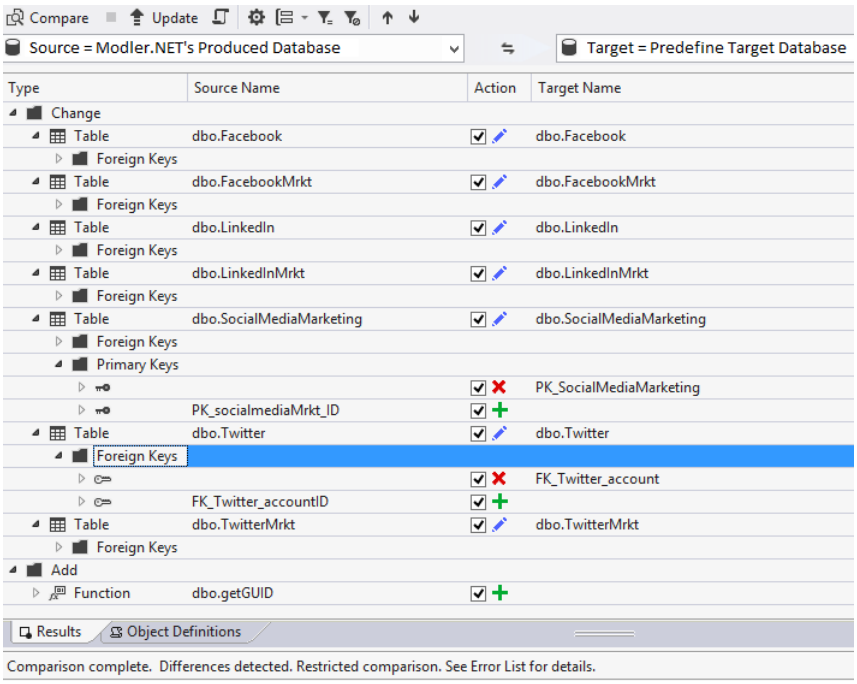

Figure 5.1: Schema Comparison Overview - Non-Structural Modifications Differences

Based On	Model Name	Nr. of Entities	Nr. of Attributes	Nr. of Links
Input Data Set - 1				
IM $_1$	TM-struct $_1$	48	2943	49
IM $_1$	TM-nonStruct $_1$	49	2933	50
IM $_1$	TM-complex $_1$	46	2921	47
IM $_1$	TM-semantic $_1$	45	2917	46
Input Data Set - 2				
IM $_2$	TM-complex $_2$	44	2856	49
IM $_3$	TM-complex $_3$	36	2322	38
IM $_4$	TM-complex $_4$	40	2704	40

Table 5.3: Overview - Data Model Input Sets

Transformation / Operation	Model		Database	
	Achieved	Not achieved	Achieved	Not achieved
Entity Creation	x		x	
Entity Removal	x		x	
Attribute Creation	x		x	
Attribute Removal	x		x	
Relationship Creation	x		x	
Relationship Removal	x		x	

Table 5.4: Structural Modifications Overview

The figure shows that the produced database contains an extra function called 'dboGetGUID'. This function is used by Modler.NET to convert column data types while ensuring that the related data values are also converted. Furthermore, the naming of foreign-key constraints is different (cf. details in listing 5.1). Instead of the 'referencedTable_referencingTable' convention used by SQL Server 2012, the tool uses a 'referencedTable_referencingAttributeInReferencingTable' naming convention.

```
1 /* Modler.NET's way of defining FK constraints:  */
2 ALTER TABLE [dbo].[TwitterProfile]
3   ADD CONSTRAINT [FK_TwitterProfile_account_ID] FOREIGN KEY ([account_ID
      ]) REFERENCES [dbo].[account] ([AccountId]);
4 GO
5
6 /* The SQL Server Management Studio's way of defining FK constraints: */
7 ALTER TABLE [dbo].[TwitterProfile]
8   ADD CONSTRAINT [FK_Twitter_account] FOREIGN KEY ([account_ID])
      REFERENCES [dbo].[account] ([AccountId]);
9 GO
```

Listing 5.1: Schema Comparison Overview - FK Constraint Naming Difference

Table 5.6 shows that the tool's database transformations for most of the *complex modifications* did not fully meet the equivalence criteria. They failed to produce the correct migrated data values.

Extracting attributes from one entity to another entity with which it has no relationship,

Target Equivalence / Operation	Model		Database	
	Achieved	Not achieved	Achieved	Not achieved
Entity Rename	x		x	
Attribute Rename	x		x	
Attribute Change DataType	x		x	
Attribute Make PrimaryKey	x		x	
Attribute Drop PrimaryKey	x		x	
Super Type Entity Creation	x		x	
Super Type Entity Removal	x		x	
Composite Relationship Creation	x		x	
Composite Relationship Switch	x		x	

Table 5.5: Non-Structural Modifications Overview

Target Equivalence / Operation	Model		Database	
	Achieved	Not achieved	Achieved	Not achieved
Extract Entity	x			x
Inline Entity	x			x
Fold Entity	x			x
Unfold Entity	x			x
Move attribute over rel.	x		x	
Collect attribute over rel.	x		x	

Table 5.6: Complex Modifications Overview

Target Equivalence / Operation	Model		Database	
	Achieved	Not achieved	Achieved	Not achieved
Merge Entity	x			x
Split Entity	x			x
Merge Attribute	x		x	

Table 5.7: Semantic Modifications Overview

results in a faulty database transformation. The database transformation script attempts to insert null values into the target entity's existing columns. If one of these columns has a NOT Null constraint this will result in a database error. This error was avoided by first establishing a relationship between the entities and using it in the join-condition used to migrate the data to the target entity.

The database transformations related to the inline, fold, and unfold modifications suffer from the same shortcomings as the database transformation for entity extractions.

The merging and splitting of an entity, as shown in table 5.7, experience the same transformation errors described above.

Feasibility Observations The tool's output starts to stray from its intended target when dealing with the operators involving the exchanging of attributes. The tool produces the correct model transformation and the correct database schema transformations, but the data-value migration is incorrect. This is a result of the inconsistent input coming from the underlying VMSDK framework. This causes the event-recorder to record incomplete model changes.

In the specific case of the *merge entity* operator, the VMSDK removes the relationship data between the target entity and the source entity before the recorder can log it. A similar issue was dealt with using a custom storage mechanism, however it was not used in this instance because of time constraints. This emphasizes one of the inherent limitations of the operation-based approach that hinders its adoption, namely limitation **L5**. The success of the approach is dependent on how well its components are integrated into the chosen modeling environment. However, it can be said from experience that this integration requires a significant implementation investment, because it requires an understanding of the modeling environment's underlying framework.

Additionally, from looking at the implementation of the extract, inline, (un)fold, merge and split operations it was evident that they use similar patterns. This might be the reason for them exhibiting the same database transformation issues.

5.2.2 Case II - Expressiveness Results

As described in section 5.1.2, there are three parts to this case. The first is to determine the existence of recurring transformation patterns. The second is to divide the operators into two groups; basic operators and composable operators. The third is to show that a composition framework facilitates the expressiveness of the proposed approach.

Each of the parts are discussed separately in the following paragraphs.

Operation \ Detected Pattern	Modeling Pattern	Database Pattern
Entity Creation	(none)	(none)
Entity Removal	(none)	(none)
Attribute Creation	(none)	(none)
Attribute Removal	(none)	(none)
Relationship Creation	(none)	(none)
Relationship Removal	(none)	(none)

Table 5.8: Pattern Detection for Structural Modifications

Operation \ Detected Pattern	Modeling Pattern	Database Pattern
Entity Rename	(none)	(none)
Attribute Rename	(none)	(none)
Attribute Change DataType	(none)	Remove Attribute, Rename Attribute
Attribute Make PrimaryKey	(none)	(none)
Attribute Drop PrimaryKey	(none)	(none)
Super Type Entity Creation	Create Entity, Create Link	Create Entity, Create Link, Create Foreign-key (FK) constraint
Super Type Entity Removal	Remove Entity [4]	Remove Entity, Remove FK constraint
Composite Relationship Creation	Create Link	Create Link, Add FK constraint
Composite Relationship Switch	Remove Link, Create Link	Create Link, Remove FK constraint, Create FK constraint

Table 5.9: Pattern Detection for Non-Structural Modifications

Recurring Transformation Patterns

The results from using the code clone utility show that the *structural modification* operators do not display a use of recurring transformation patterns. The same can be said of the entity and attribute renaming operators and the creation and removal of the primary constraint operators, which are part of the *non-structural modification* operators. The remaining operators do contain recurring transformations. A detailed account of which transformation patterns each operator uses is depicted in tables 5.9 through 5.11.

Table 5.9 contains a noteworthy occurrence. The *attribute change dataType* operator displays the use of database transformation patterns but no model transformation patterns. This occurrence is caused by the fact that, at the model-level, the changing of an attribute's data type value constitutes a simple property setting through the tool's user-interface. At the database-level however it involves either an implicit or explicit data value conversion (cf. Section 4.3.4 for further details). In this particular case an explicit conversions is needed, which is handled by copying and simultaneously converting the existing data value into a temporary column that uses the new data type. The original column is then dropped and the temporary column is renamed to replace the original column. Hence, the detected attribute remove and rename patterns.

The *split entity* operator can move selected attributes in the source entity to an existing destination entity or it can move them to a new entity. This operator's use of the *create entity* pattern is therefore dependent on whether a new destination entity is needed. Hence, the asterisk behind the 'Create Entity' in table 5.11.

[4] Implies an implicit removal of the connected links

Operation \ Detected Pattern	Modeling Pattern	Database Pattern
Extract Entity	Create Entity, Create Attribute, Create Link, Remove Attribute	Create Entity, Add FK constraint, Move Attribute
Inline Entity	Remove Link, Create Attribute, Remove Attribute, Remove Entity	Remove Entity, Remove FK constraint, Move Attribute
Fold Entity	Create Attribute, Remove Attribute, Create Link	Add FK constraint, Move Attribute
Unfold Entity	Remove Link, Create Attribute, Remove Attribute	Remove FK constraint, Move Attribute
Move attribute over rel.	Create Attribute, Remove Attribute	Move Attribute
Collect attribute over rel.	Create Attribute, Remove Attribute	Move Attribute

Table 5.10: Pattern Detection for Complex Modifications

Operation \ Detected Pattern	Modeling Pattern	Database Pattern
Merge Entity	Create Attribute, Remove Attribute	Move Attribute
Split Entity	Create Entity*, Create Attribute, Remove Attribute	Create Entity*, Move Attribute
Merge Attribute	Create Attribute, Remove Attribute, Remove Entity	Move Attribute, Remove Entity

Table 5.11: Pattern Detection for Semantic Modifications

Operator Grouping

Based on the results from tables 5.8 - 5.11 it is evident which operators were basic operators and which were composable operators. The following is an overview of all the operators per group.

Basic Operators:

- Entity creation, entity removal and entity renaming
- Attribute creation, attribute removal, attribute renaming, attribute data type changing and attribute moving
- Setting and removing primary-key or foreign-key constraints on an attribute

Composable Operators:

- Super-type creation and its removal
- Composite Relation creation and switching
- Entity extraction, inlining, folding, unfolding and splitting
- Merging of entities or attributes

Expressivity of Operator Composition

In this paragraph the expressivity of the operator composition framework is assessed by tackling three characteristics of expressiveness: conciseness, precision and completeness.

Starting with the conciseness characteristic of the composition framework, the following sub-section shows that by re-implementing the composable operators as composite operators, an average reduction was achieved of 79%. An overview of the averaged reduction results per modification type is given in Figure 5.2 and the detailed breakdown of the reduction results per operator is given in appendix C.

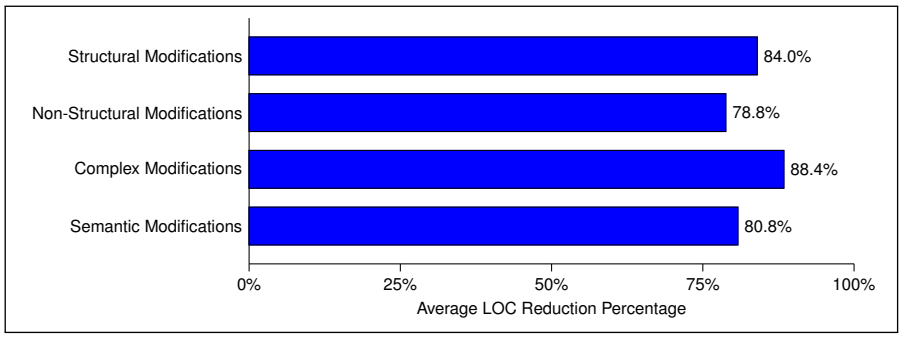

Figure 5.2: Lines-Of-Code Reduction Overview

As for the precision of the operator implementations, the following code listings show the comparison between the composable operators and their composite counter-parts. These composite counter-parts were created by following the implementation details of the operator they are replicating. The main differences between the two operator definitions are that the composition framework abstracts over the transaction-related statements and the object instantiation statements. These statements are handled internally so that the needed input are the object's property values.

The first operation that displayed a use of both model transformation patterns and database transformation patterns is the *SuperType Entity Creation* operator. It was shown to use two basic operators; *entity creation* and *link creation*. Listing 5.2 and 5.3 depicts the operator's implementation as a composition operator and as a composite operator. The use of abstraction and operator functions in listing 5.3 seems to result in a more straightforward (i.e. precise) operator implementation.

```
using (t = transactionMngr.BeginTransaction("Create Super Type"))
{
  entity = new Entity(store);
  entity.ModelRoot = eInfo.modelRoot;
  entity.Name = eInfo.name;
  entity.IsImported = eInfo.isImported;

  foreach (Entity ent in entityList)
  {
    relationship = new EntityAssociation(entity, ent);
    relationship.AttributeInSource = "";
    relationship.AttributeInTarget = "";
    attributeInTarget.isForeignKey = true;
  }
  t.Commit();
}
```

Listing 5.2: Operator - Create Super Type

```
Operator op = Operators.Define(store);
op.CreateEntityOp(modelRoot, entityName, isImported, out
    newEntityID, null);

foreach (Entity ent in entityList)
  op.CreateLinkOp(newEntityID, ent, "", attributeInTarget, false,
      true);
```

Listing 5.3: Equivalent Composite Operator

The next operator, *SuperType Entity Removal*, is compared to its composite counter-part in listing 5.4 and 5.5.

```
using (t = transactionMngr.BeginTransaction("Remove Super Type"))
{
  foreach (Entity ent in entity.Targets)
    ent.EntityAttributes.Where(a => a.isForeignKey == true && a.
        ReferencedEntityID == entity.Id).ToList().ForEach(aa => aa
        .isForeignKey = false);
  entity.Delete();
  t.Commit();
}
```

Listing 5.4: Operator - Remove Super Type

```
1 Operators.Operator ops = Operators.Define(store);
2 foreach (Entity ent in entity.Targets)
3   ops = ops.UpdateAttributeProperty(EntityAttribute.
        isForeignKeyDomainPropertyId, ent.EntityAttributes.Where(a
        => a.isForeignKey == true && a.ReferencedEntityID == entity.
        Id).ToList(), false, ent);
4
5 ops.RemoveOperator(new List<Entity> { entity });
```

Listing 5.5: Equivalent Composite Operator

The two composite relationship operators have a similar build up to the superType operators described above (cf. B.2).

The *extract entity* operator is similar to the entity inline, fold, unfold, merge and split operators. They all have the same basic make-up. Listing 5.6 and 5.7 show how such an operator was reproduced using composition.

```
1  using (t = transactionMngr.BeginTransaction("Create Relationship
       For Extracted Entity"))
2  {
3    targetEntity = new Entity(store);
4    targetEntity.ModelRoot = modelRoot;
5    targetEntity.Name = name;
6    targetEntity.IsImported = isImported;
7
8    relationship = new EntityAssociation(sourceEntity, targetEntity);
9    relationship.AttributeInSource = "";
10   relationship.AttributeInTarget = "";
11   attributeInTarget.isForeignKey = true;
12   t.Commit();
13 }
14 using (t = transactionMngr.BeginTransaction("Move Attributes"))
15 {
16   foreach (EntityAttribute attr in attributeList)
17   {
18     targetEntity.EntityAttributes.Add(attribute.Copy() as
         EntityAttribute);
19     sourceEntity.EntityAttributes.Remove(attribute);
20   }
21   t.Context.Add(Operators.GetMoveOpTransactionKey, sourceEntity);
22   t.Commit();
23 }
```

Listing 5.6: Operator - Extract Entity

```
1 Operator op = Operators.Define(store);
2 op.CreateEntityOp(modelRoot, name, isImported, out newEntityID,
      null);
3
4 op.CreateLinkOp(sourceEntity, newEntityID, "", attributeInTarget,
      false, true).MoveOperator(attributeList, newEntityID);
```

Listing 5.7: Equivalent Composite Operator

Based on these listings it can be said that the operator composition framework results in a more concise and precise implementation code.

Regarding the completeness of the framework, there are three facts that can be used to establish that the framework possesses this characteristic. The first is that it was possible to re-implement all of the operators from the operator list using the framework. Secondly, that list of operators was researched and concluded to be practically-complete [39]. As a result of that, it can be said that the framework is able to express all probable co-evolution tasks. Thirdly, the framework is based on the C# programming language which is Turing-Complete (i.e. maximally expressive). Since the expressiveness of the syntactic mechanism of a "host" language is inherited by that which is built on it [48], the framework's syntax allows for maximum expressivity. Thus, the operator composition framework is complete in terms of its ability to define operators.

Expressivity Observations While conducting the experiment a few key observations were made. The first was that, the addition and removal of foreign-key (FK) constraints are not explicitly listed as an operator, even though the addition and removal of primary-key constraints are listed. The FK constraint creation and removal operation is used in 8 of the 13 composable operators. Because it is so frequently used it was added as a basic operator, so that it can be used to compose new operators.

Secondly, it was observed that the use of operator composition inherently enables the reuse of transformation knowledge when creating operators. This reuse was the main contributing factor for the LOC reduction that was achieved for the composable operators. Meaning, the LOC reduction for the composable operators is primarily dependent on the reuse of the transformation code for the basic operators.

Thirdly, it was shown that the new framework increases the expressiveness with which operators can be defined by facilitating the use of operator implementation code that is more concise and precise while remaining flexible enough to cover arbitrary co-evolution scenarios.

There are also a few drawbacks. There is a complexity increase from just navigating the operator library to include the complexity of deciding what the correct parameters are for an operator function. Additionally, the more expressive a language is the more complex it will be to analyze. Furthermore, the higher level of abstraction may hinder expressiveness in terms of not allowing specificity, hence the need for mixing the operation invocations with custom statements when defining some of the composite operators.

5.2.3 Case III - Adaptability Results

This case is centered around the OP-DEVO approach's ability to retroactively adapt database transformations. The component of Modler.NET that was used in this case was the EventManager utility (cf. Section 4.4.5) which is built on top of the event-based change recorder. This utility facilitates the process of locating an event as well as the process of adapting the generated database transformation code that is linked to that model modification event.

The transformations and history logs from Case I were used as the source for the inputs. Specifically, the scripts generated from the evaluations of the complex and semantic model modification types. The tool was shown to produce incorrect database transformations when reproducing these modification types. Thus, they represent actual scripts that need correction. Furthermore, it was stated in Case I that the complex and semantic operators produce an incorrect database transformation because of the modification order issue. The ordering of a model modification is an example of a situation where the modifications at the model-level are a direct cause of incorrect database transformations.

The following is a detailed outline of this adaptation being performed on target model TM-complex $_1$.

Adaptation Outline For TM-complex $_1$

It is known that the transformation error related to the *extract entity* operator is caused by the attribute move event that could not establish a relationship between the source entity "account.accountid" and the target entity "addressInfo.account_ID". Using the EventManager this can be corrected by importing all of the primary-key values into the target entity (cf. 5.8). This was added to the SQL-mapping value for the *foreign-key creation* event between the two entities. Once this is done, the join-condition " *WHERE accountid = [AddressInfo].account_ID*" is no longer NULL, thus allowing the UPDATE statement to find a set of rows to update.

After this adjustment has been made, the updated database transformation script is obtained by simply re-generating the script.

```
1 INSERT INTO [dbo].AddressInfo (account_ID)
2   SELECT DISTINCT accountID FROM [dbo].account
3 GO
```

Listing 5.8: Import PK Values - Correcting Join Condition

Listings E.1 and E.2, which are located in Appendix E, show the incorrect database transformation and the adjusted transformation for the extract operation.

Adaptability Observations By using the EventManager it is fairly easy to pin-point which database transformation statements correspond to a particular model modification. When a recorded model modification is selected, it displays the corresponding database transformation in an editable textbox.

Furthermore, any database transformation that was generated from a model modification can be adjusted at any point during the model evolution process by using the EventManager and that the adjustment can be done independently. This means that it is possible to decouple the database transformations from the model modifications that it was generated from.

Thus, erroneous database transformations can be retroactively corrected without having to redo any model modifications or having it affect other existing database transformation statements.

5.3 Validity Threats

Despite the efforts made to maintain the integrity of these case studies, there are aspects of the chosen evaluation methodology that can negatively impact the integrity of the obtained results and thereby impact the validity of the project itself. These validity threats are described below.

5.3.1 Internal Threats

The main internal threat was the unpredictable behavior of the modeling platform that was used. The more functionality that was added to Modler.NET, the more unpredictable the platform became. This problem expressed itself as inconsistent communication with the change recorder which was tightly integrated with it. This threatened the validity of the obtained results for Case I and III. Furthermore, the lack of maturity and correctness of the tool's implementation could decrease the validity of the results even more.

To minimize the threat of the modeling platform and the tool, the tool's implementation and its produced results were tested individually. The implementation code was tested by writing tests. This lessens the probability of the tool itself being the cause of systematic errors. In order to address the modeling platform threat, the results produced by the tool were continuously tested. This was done by investigating each incorrect result. The intent of these result-checks was to pin-point the cause. If the error was caused by a bug in the implementation, this was corrected and the task was repeated. Otherwise, it was considered a valid transformation that shows that the tool does not support that particular transformation.

5.3.2 External Threats

As stated in Section 5.1 the data models that were used as input were generated from CRM meta-data. The choice was made to use this type of input because data model and database co-evolution is an actual recurring scenario in the development and maintenance of CRM applications, the data in the CRM application is realistic, and generating models instead of creating them by hand saves time. This means that the selected input is too small and specific to be a viable representative of graphical data models in general. Therefore, it is not possible to claim that the obtained results are valid for all graphical data model development. That which can be confidently claimed, however, is that the results are valid for CRM-based

data models. This makes the results valuable and suitable to those whose work revolves around the management and development of CRM and CRM-related applications.

5.3.3 Construct Threats

The evaluation was based on various comparisons between models and databases. In Case I and III the equivalence measurement of the database pairs was done using Visual Studio's 'Schema Compare' tool. The model pairs in Case I were compared by hand. In Case II the existence of recurring patterns was determined using Visual Studio's 'Code Clone Analyser'. If these measurements of the model equivalence and database equivalence were to be incorrect, that error - whether it be accidental or systematic - would skew the results. In order to minimize this threat the obtained results were manually double checked. If an inconsistency was detected, it was manually corrected. In the case of detecting patterns using the code clone utility, the tool was not as accurate as was hoped. It required significant manual correction to find and log all viable patterns.

5.4 Results Summary

Despite the shortcomings of the tool's recorder, the results of the feasibility assessment show that the proposed approach could support the different types of model modifications as well as generate viable database transformations from these model changes. In other words, Modler.NET is capable of supporting arbitrary model transformations and, provided a more thorough recorder implementation, the database transformation that is generated from these model modifications can produce are a viable migration script.

The expressiveness case study revealed that the use of the operator composition framework does increase the expressiveness with which new/custom operators are defined. This increased expressiveness leads to a decrease in operator development effort by providing an average LOC reduction of 79%. Consequently, there is a potential reduction in operator implementation-time as well. The LOC reduction is primarily dependent on the fact the the framework allows for reuse of existing operators. Thus, the framework allows for a more concise and precise definition of custom operators. Furthermore, because the modeling language's host language is C#, the framework can also provide the needed flexibility to cover arbitrary transformation scenarios.

As for the adaptability assessment, it was shown that it is possible to quickly pin-point a database transformation from the list of recorded model modifications. Moreover, individual event data can be retroactively adjusted after which the transformation script can be regenerated to include these changes. This indicates that the proposed approach supports independent adaptation of its ModelTransformation-2-SQLStatements mappings on-the-fly (i.e. at any point during the model evolution process).

Chapter 6

Conclusions and Future Work

Applications evolve over the course of time, therefore, their data models need to be adapted. To reduce the effort required for data model and database co-evolution, an automation solution is needed that can transform a database in such a way that it can store the new data, that conforms to the evolved data model, as well as retain the existing data.

Chapter 2 gave an overview of the related work being done in this field. It states that there are two main approaches for automating model co-evolution, namely: the model matching-based approach and the operation-based approach. When dealing with data model with a graphical notation, the operation-based approach has two advantages over its model matching-based counter-part. The first is that it tracks model changes more accurately, thereby, facilitating the production of correct database co-transformations. Secondly, its model change tracking technique is more accurate and it is less computationally complex. Thus, it produces better results in less time [44, 32, 41]. Consequently, the operation-based approach seems to be the better choice, but it has some inherent limitations. These limitations were described in chapter 3. After having analyzed these limitations it became evident that the standard operation-based approach could be improved upon. Therefore a redesign was proposed that aims to improve the approach's expressiveness and adaptability through:

- ◇ Operator composition which should mitigate the expressivity limitations that complicate the creation of custom transformations
- ◇ Event-sourcing-based change recording which should facilitate on-the-fly adaptation of the generated database transformations

6.1 Conclusion

The goal of this research project was to redesign the operation-based approach so that it could provide better support for the coupled evolution of graphical data models and their data. This improved support was to be obtained by addressing the approach's inherent limitations related to its expressiveness and its adaptability. The main research question, therefore, was: *"How can the limitations inherent to the operation-based approach for automating the co-evolution of data models with graphical notation be mitigated by using*

operator composition and event-sourcing patterns?". This question was divided into three sub-questions which were used to guide the research. This section provides the answers to all of the questions, starting with the sub-questions, and ending with the researcher's conclusion.

RQ1: *What are the limitations inherent to the operation-based approach to automating the co-evolution of data models with a graphical notation?*

As mentioned in chapter 3, the operation-based approach has five inherent limitations that can be grouped into three categories. The first is *Coupled Operator Limitations* which consists of three of the five limitations, namely:

L1 - It is favorable to have a rich operator library because the amount of model modification types that an approach can support is related to the amount of operators in its library. However, the larger the operator library is, the harder it is to find the correct operator for a particular task. In other words, there is a tradeoff between the size of the operator library and the ease of finding the correct operator

L2 - From **L1** it can be deduced that having a operator library with a fix set of operators means that there are a fixed amount of model modification that such the approach can support. Therefore, it would be favorable to allow the creation of new operators instead of using a fixed operator library

L3 - In order to allow the creation of custom operators to expand the operator library, the approach has to provide the developers with a modeling language that is expressive enough for them to create arbitrary operators. These modeling languages do not capture migration patterns or other reoccurring patterns, therefore the new operators do not benefit from reusing the existing transformation knowledge

The second category is called the *Database Co-Evolution Limitation*, which only has one limitation (**L4**). This limitation has to do with the fact that the coupling of database migration code to a model transformation impedes independent adaptation of either the model or the database migration code. This implies that the developer needs to take the impact of the data model changes on the database into account. Since the generated database transformations are based on the recorded model modifications, the order of the model modification can affect the generated database transformation. If a modification order leads to an incorrect database transformation this would mean that the performed model modifications will have to be undone and then repeated in a different order to fix the issue. Therefore, it would be beneficial to know which modifications caused the erroneous database transformation so that only those modifications can be repeated.

The last is the *Tooling Limitation* (**L5**) which is that the use of an operation-based approach requires one to have a modeling tool that implements operator library, the

model change recorder to register the model modifications, and the migration functionality responsible for the co-evolution process. Thus, it requires a significant upfront implementation investment

RQ2: *How can the use of operator composition reduce the effort required to define custom operators?*

It was shown that the new operator composition framework decreased the effort associated with the definition of a new operator by increasing the expressiveness of the operator definition language. Expressiveness was defined in terms of three aspects: the conciseness, the preciseness and the completeness (i.e. being flexible enough to support arbitrary transformations) of the definition language.

The operator composition framework defines an operator by using a function declaration that encapsulates the needed model transformation statements. An operator is, therefore, represented by a function. These functions can contain individual model transformation statements as well as existing operators. In other words, these functions facilitate operator reuse. Because of this reuse facility the implementation code needed to define custom operators is more concise and precise. This was demonstrated in the evaluation which showed an average reduction of 79% in Lines-Of-Code (LOC) needed to define non-basic operators with the new framework. The basic operators are those that are responsible for atomic transformations such as: 1) the creation or removal of an entity, attribute or reference 2) the renaming of an entity or attribute and 3) the modification of attribute properties.

The evaluation also showed that because the modeling language uses a general-purpose language (C#) as its host language, the operator composition framework is complete in terms of being able to support any co-evolution scenario.

Thus, the new operator composition framework provides a means to define new operators more concisely and precisely which reduces the coding-related effort by an average of 79%, while remaining flexible enough to cater to arbitrary transformations scenarios.

RQ3: *How can the event-sourcing pattern be used to manage the process required to map data model adaptations to a database transformation?*

The event-sourcing pattern was used to design an event-based change recorder that provides the necessary framework upon which functionality was built to manage the generation of data transformations such as: enabling one to have more control over the recorded data. Control is defined as the ability to identify specific recorded model transformations and adapting the corresponding database transformation without having it affect the existing data model.

The evaluation also showed that the management functionality, EventManager, allows erroneous database transformations to be retroactively corrected without having to repeat any model modifications or having its adjustment affect other recorded

transformations. In other words, the event-sourcing based change recorder provided a means to de-couple the database transformations from the model modifications that it was generated from.

Thus, the new event-based change recorder enables one to manage the processes pertaining to database transformation mapping by: 1) providing a platform which records model modifications as a detailed event 2) facilitating search and adaptation functionality of those events and 3) this can all be done at any point-in-time during the model evolution process.

RQ $_{\text{main}}$: *How can the limitations inherent to the operation-based approach for automating the co-evolution of data models with graphical notation be mitigated by using operator composition and event-sourcing patterns?*

The answer to the main question is found by combining the previous answers as follows.

The operator composition framework deals with limitations **L2** and **L3** in the following manner. The proposed approach provides the user with an operator composition framework with which new/custom operators can be created. As for the knowledge reuse, the operator composition framework inherently enables the reuse of existing modeling patterns to define new operators. Furthermore, it is also possible to reuse the basic operators as well as the composite operators to create other operators. The composition framework is built on top of the standard modeling environment, therefore it is also possible to write model transformation code in C#. This gives the composition framework the flexibility needed to define arbitrary model transformation.

The event-based change recorder addresses limitation **L4** by enabling control over data transformations. This control will not correct the ordering issue itself, but rather it will allow the related transformations to be identified and corrected. The approach utilizes an asynchronous database transformation process. Meaning, the transformation is not executed immediately. This makes it possible to adjust the transformation before it is executed. Therefore, the generated transformation can be checked at any point in time during the model evolution and if an error is detected in the database transformation - whether it be from the ordering of an adaptation or not - it can be dealt with as follows. First, the event logs are used to pinpoint the event that caused the error. The error in the event is corrected by either adjusting the SQL statements generated from the recorded model change, or setting the doSkip-flag which will cause the script generator to not add this event's SQL statements to the transformation script. Once the incorrect event is fixed, the script generator is used to regenerate the transformation script. Thus, erroneous database transformations can be retroactively corrected without having to redo any model modifications.

As for limitation **L1**, Modler.NET uses context-aware context-menus which provide the user with a list of possible operators that can be applied to the selected model element(s). This list is filtered based on the selected element(s), therefore only those

operators that are relevant are displayed. This use of context-awareness reduces the selection complexity issue by filtering the whole operator library down to only the relevant operators and thereby making it easier to select the required operator.

The last limitation, **L5**, was addressed by making the tool that was created to evaluate the proposed approach, Modler.NET, available online. The core functionality of the recorder and operators will be made available at `https://modlernet.codeplex.com/`, thus providing an alternative to building the operator library and recorder from scratch. This will reduce the upfront investment needed to get started with operation-based data model co-evolution.

The end result of this research project is an operation-based approach that has mitigated its expressivity and its adaptability limitations by incorporating two components. The first is an operator composition framework which was shown to increase the approaches expressiveness in defining new operators by an average of 79%. The second component was the event-based change recorder which logged each individual model modification as an event. These events contained all the data necessary to be able to manage the generated contents of a database transformation script. This meant that the script could be modified separately from the data model at any time, thus increasing the adaptability of the generated database transformations. In addition to the aforementioned limitations, the proposed approach also addressed two other limitations, namely **L1** and **L5**.

Based on these results, it can be concluded that this research project achieved its goal.

6.2 Contribution

The contribution of this thesis project can be summarized as the redesign of an operation-based approach that, in addition to supporting the coupled evolution of data models with graphical notation by coupling data model modifications to database migration, also aims to address the inherent limitation of the operation-based approach through the use of *operator composition* and *event-based change recording*. Furthermore, the tool (i.e. Modler.NET) [1] that was developed based on this approach and the case studies which were used to evaluate it, can be used as resources to further the research on graphical data model co-evolution.

6.3 Discussion

The solution presented in this research project, as with any project, has its pros and cons. These properties of the end result are discussed in the following sections.

6.3.1 Advantages of the Proposed Redesign

The OP-DEVO approach adds two components to the standard operation-based approach; the operation composition framework and the event-based recorder. The operation compo-

[1] Available at: https://modlernet.codeplex.com/

sition framework enables reuse of existing transformation knowledge which was shown to lead to increased expressivity. These added properties facilitate the development of new co-evolution operation by providing a modeling environment which takes advantage of existing transformations - which have already been tested and therefore are likely to have less bugs than newly written transformations - and of a simpler modeling language that allows the new operator to be defined as a sequence of functions instead of lines of model and database transformation code.

Furthermore, the OP-DEVO approach can be used as a platform on which to build tools, similar to Modler.NET, that visualize model modifications and their impact on the database schema and data. This is made possible by the change recorder. The level of detail with which each model modification can be recorded and the readily available SQL mapping for those modifications means that it can be a valuable resource for further database migration/evolution research such as visualization aids for schema evolution (see future works section 6.4).

6.3.2 Drawbacks of the Proposed Redesign

The use of the operator composition framework adds to the operator selection complexity. The visual editor window has a context-menu which provides the user with a list of possible operators that can be applied to the selected model element(s). That list is filtered based on the selected element(s). When programming a new operator, however, it is difficult to determine which operator needs to be filtered out because there is no context information. The operators themselves are not aware of the elements on which they will be applied. Therefore, one can define a new operator that uses the 'remove-operator' to remove an entity followed by a 'rename-operator' that renames the entity which was deleted. In other words, there needs to be a mechanism by which the programming of new operators can be semantically checked.

Furthermore, there is a learning curve related to the use of the operator composition framework. Most of the operators have multiple instances each having its own set of parameters. Thus, one needs to know which of the instances are available and figure out which parameter can be filled in based on the available model information.

The granularity of the recorded model modifications can also be problematic, in that it is not able to group relevant statements together. This leads to inefficient database transformations. An example is the simple action of removing an entity which can be done by just dropping the related table, but because of the granularity of the recorder the delete-action is translated as the deletion of individual columns and then the table itself is dropped. A possible solution for this would be to provide extra information in the recorded events that would allow the script generator to know which data to group and how to group them.

Lastly, the conclusions are based on the results of the evaluation of which the selected input (i.e. CRM meta data) is too small and specific to be a viable representative of graphical data models in general. Therefore, the results may only be valuable and suitable to those whose work revolves around the management and development of CRM and CRM-related applications.

6.4 Future Work

While conducting this research a number of ideas surfaced on how the proposed approach could be improved, as well as an idea to compare the redesign approach to its difference-based counter-part. The following subsections outline these potential future research topics.

6.4.1 Multi-User Support

The use of model-driven development in industry has been increasing, hence the need for managing these models in terms of change tracking and versioning. Version control is a well-known solution for supporting multi-users for textual artifacts such as source code. There is however very little being done for graphical artifacts such as the models dealt with in this thesis [44, 31]. Thus, the integration of multi-user support in the form of version control could be a potential research area worth exploring.

The proposed approach could potentially use a textual Version Control Software (VCS) to track both the model evolution and the database transformation. The first step would be to save the initial data model and the database. Secondly, use the textual VCS to track the recorded change-events.

If a particular version of the data model is needed it can be obtained by de-serializing that version of the recorded events and applying it to the initial model which will in effect produce the required version of the data model. Furthermore, it would be possible to produce the corresponding database by generating a transformation script from the recorded events and applying it to the saved database.

6.4.2 Interactive Database Migration

Interactive database migration entails the use of a migration algorithm that automatically migrates the database in the background, and whenever it needs supplementary information, it asks the model developer to provide the missing information [38].

This interactive approach to the migration process will enable syntactic and semantic checking of the generated SQL statements (i.e. ensure statement validation and correctness). Consequently, this will allow the user to fix any database-related errors from within the model development environment.

6.4.3 Difference-based vs. Operation-based

There has yet to be a research project in which, given the same co-evolution scenario, the difference-based approach is compared to the operation-based approach using quantitative measures. None of the papers that were read during the conducted literature survey [41] or those read during this project give actual numbers or percentages when comparing the two approaches. Hence, there should be a research project that can quantify the ability of these approaches to perform the same specific custom data model co-evolution tasks. The 'ability' of each approach should be defined in terms of *accuracy* (the exactness of the recorded model evolution), *correctness* (the production of a valid database co-evolution that handles existing and future data properly) and *time to execute* (the time it took to apply the

specified co-evolution tasks and yield an adequate result). The results should be compared and used to deduce which approach should be used in the given scenarios.

Hopefully, such a research will lead to insights on any correlation between the choice of approach and the co-evolution scenarios or the data model types. If such a correlation exists, then the research can be used to determine - based on concrete data instead of theoretical arguments or preference - which approach should be used when dealing with a specific scenario or data model type.

Bibliography

[1] M.A. Aboulsamh and J. Davies. A metamodel-based approach to information systems evolution and data migration. *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, pages 155–161, 2010. ICSEA '10.

[2] M. Alanen and I. Porres. Difference and union of models. *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, 2863:2–17, 2003. Lecture Notes in Computer Science.

[3] T. Alves, P.F. Silva, and J. Visser. Constraint-aware schema transformation. *The Ninth International Workshop on Rule-Based Programming*, pages 16–96, 2008.

[4] S.W. Ambler and P.J. Sadalage. *Refactoring databases: evolutionary database design*. The Addison-Wesley signature series. Addison Wesley, 2006.

[5] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for xml and sql. *Practical Aspects of Declarative Languages*, pages 290–304, 2007.

[6] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[7] C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European J. for the Informatics Professional*, 2008. April-May 2008.

[8] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. *IEEE Computer Society - Enterprise Distributed Object Computing Conference*, pages 222–231, 2008.

[9] A. Cicchetti, D. Di Ruscio, L. Iovino, and A. Pierantonio. Managing the evolution of data-intensive web applications by model-driven techniques. *Software and Systems Modeling*, pages 1–31, 2011. (16 February 2011).

[10] A. Cleve, AF. Brogneaux, and JL. Hainaut. A conceptual approach to database applications evolution. *Conceptual Modeling–ER 2010*, pages 132–145, 2010.

[11] A. Cleve and J.L. Hainaut. Co-transformations in database applications evolution. *Generative and Transformational Techniques in Software Engineering*, 4143:409–421, 2006. Lecture Notes in Computer Science.

[12] S. Cook, G. Jones, S. Kent, and A.C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.

[13] A. Cunha, J. Oliveira, and J. Visser. Type-safe two-level data transformation. *FM 2006: Formal Methods*, pages 284–299, 2006.

[14] K. Czarnecki and S. Helsen. Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[15] K. Czarnecki and S. Helsen. Feature-base survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[16] A. Van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. *CSMR Workshop on Model-Driven Software Evolution (MoDSE'07)*, pages 41–49, 2007.

[17] E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Medea: A database evolution architecture with traceability. *Data & Knowledge Engineering*, 65(3):419–441, 2008.

[18] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman, 2006. Boston, Massachusetts, 5th edition.

[19] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, 5301:326–340, 2008. MoDELS '08.

[20] M. Fayad and M.P. Cline. Aspects of software adaptability. *Communications of the ACM*, 39(10):58–59, 1996.

[21] The Eclipse Foundation. Eclipse modeling framework project (emf). http://eclipse.org/modeling/emf/, 2013.

[22] M. Fowler. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html, 2013.

[23] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. *Proceedings of European Conference on Modelling Foundations and Applications (ECMDA-FA)*, 5562:34–49, 2009. LNCS.

[24] B. Gruschko, D. Kolovos, and R. Paige. Towards synchronizing models with evolving metamodels. *Proceedings of the International Workshop on Model-Driven Software Evolution*, 2007.

[25] G. Guerrini, M. Mesiti, and D. Rossi. Impact of xml schema evolution on valid documents. *Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, 2005.

[26] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993. ACM.

[27] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45:451–461, 2006.

[28] T. Halpin and T. Morgan. *Information Modeling and Relational Databases*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2010.

[29] M. Hartung, J. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. *Schema Matching and Mapping*, pages 149–190, 2011.

[30] M. Herrmannsdörfer. Metamodels and models, 2007. Master's Thesis, M ünchen University of technology, München, Germany, July.

[31] M. Herrmannsdörfer. Operation-based versioning of metamodels with cope. *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 49–54, 2009.

[32] M. Herrmannsdörfer. Cope a workbench for the coupled evolution of metamodels and models. *Software Language Engineering*, 6563:286–295, 2011.

[33] M. Herrmannsdörfer, S. Benz, and E. Juergens. Automatability of coupled evolution of metamodels and models in practice. *Model Driven Engineering Languages and Systems*, 5301:645–659, 2008. Lecture Notes in Computer Science.

[34] M. Herrmannsdörfer, S. Benz, and E. Juergens. Cope: A language for the coupled evolution of metamodels and models. *In Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[35] M. Herrmannsdörfer, S. Benz, and E. Juergens. Cope: Coupled evolution of metamodels and models for the eclipse modeling framework. *Eclipse Modeling Symposium*, 2008.

[36] M. Herrmannsdörfer, S. Benz, and E. Juergens. Cope - automating coupled evolution of metamodels and models. *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 52–76, 2009. Berlin, Heidelberg.

[37] M. Herrmannsdörfer and M. Koegel. Towards a generic operation recorder for model evolution. *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 76–81, 2010.

[38] M. Herrmannsdörfer and D. Ratiu. Limitations of automating model migration in response to metamodel adaptation. *Models in Software Engineering*, 2010.

[39] M. Herrmannsdörfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. *Software Language Engineering, Third International Conference (SLE 2011)*, 2011. Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers.

[40] J-M. Hick and J-L. Hainaut. Strategy for database application evolution: The db-main approach. In *Conceptual Modeling-ER 2003*, pages 291–306. Springer, 2003.

[41] P. Hunte. Managing coupled transformation of models and application data. (unpublished) Literature Survey - Delft, The Netherlands, 2012.

[42] L. C. L. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, 2010.

[43] S. Kent. Model driven engineering. *Integrated Formal Methods*, 2335:286–298, 2002. Lecture Notes in Computer Science.

[44] M. Koegel, M. Herrmannsdörfer, J. Helming, and Y. Li. State-based vs. operation-based change tracking. *Proceedings of the Joint ModSE-MCCM Workshop on Models and Evolution*, 2009. ModSEMCCM'09.

[45] M. Koegel, M. Herrmannsdörfer, L. Yang, J. Helming, and J. David. Comparing state- and operation-based change tracking on models. In *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*, pages 163–172, 2010.

[46] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 146–157, 2009. ECMDA-FA 09.

[47] D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F Paige. Different models for model matching: an analysis of approaches to support model differencing. *IEEE Computer Society*, pages 1–6, 2009. 2009 ICSEWorkshop on Comparison and Versioning of Software Models.

[48] T. Kosar, P.A Barrientos, M. Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[49] R. Lämmel. Coupled software transformations (ext. abstract). *Proceedings of the 1st International Workshop on Software Evolution Transformations*, 2004.

[50] R. Lämmel. Transformations everywhere. *Science of Computer Programming*, 52(1-3):1–8, 2004.

[51] R. Lämmel and W. Lohmann. Format evolution. *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, 155:113–134, 2001.

[52] K.K Lau and Z. Wang. A taxonomy of software component models. *Software Engineering and Advanced Applications*, pages 88–95, 2005. 31st EUROMICRO Conference on.

[53] MSDN Library. Extend your dsl by using mef. http://msdn.microsoft.com/en-us/library/ff972471, 2013.

[54] S.J. Mellor, A.N. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20(5):14–18, 2003.

[55] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, 2002. ICDE '02.

[56] MetaCase. Metaedit+ modeler - supports your modeling language. http://www.metacase.com/mep/, 2013.

[57] MetaCase. Metaedit+ workbench - build your own domain-specific modeling language. http://www.metacase.com/mwb/, 2013.

[58] Object Management Group (OMG). Mof qvt final adopted specification, 2007. March.

[59] R. Pérez-Castillo, I. García-Rodríguez de Guzmán, I. Caballero, and M. Piattini. Software modernization by recovering web services from legacy databases. *Journal of Software: Evolution and Process*, 2012.

[60] P. Ponniah. *Data Modeling Fundamentals: A Practical Guide for IT Professionals*. Wiley, 2007.

[61] R.C. Read and D.G. Corneil. The graph isomorphism disease. *J. Graph Theory*, 1(4):339–363, 2006.

[62] R. Reddy, R. France, F. Fleuery, and B. Baudry. Model composition - a signature based approach. *Proceedings Aspect Oriented Modeling workshop*, 2005. MODELS/UML 2005, Montego.

[63] L. Rose, M. Herrmannsdörfer, J. Williams, D. Kolovos, K. Garcés, R. Paige, and F. Polack. A comparison of model migration tools. *Model Driven Engineering Languages and Systems*, pages 61–75, 2010.

[64] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A Polack. An analysis of approaches to model migration. *Models and Evolution*, 2009. MoDSE-MCCM Workshop 09.

[65] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. February 2006.

[66] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Data Semantics IV*, 3730:146–171, 2005. Lecture Notes in Computer Science.

[67] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3):291–307, 2004.

[68] H. Su, D. Kramer, L. Chen, K. Claypool, and E.A. Rundensteiner. Xem: Managing the evolution of xml documents. *Research Issues in Data Engineering, 2001. Proceedings. Eleventh International Workshop on*, pages 103–110, 2001.

[69] S. Vermolen. Software language evolution. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 323 –326, 2008.

[70] S. D. Vermolen, G. Wachsmuth, and E. Visser. Generating database migrations for evolving web applications. *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE '11) ACM*, pages 83–92, 2011. New York, NY, USA.

[71] S. D. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing complex metamodel evolution. *Software Language Engineering*, pages 201–221, 2012. Revised Selected Papers, 4th International Conference.

[72] S.D. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. *LNCS*, 5301:630–644, 2008. MODELS 2008. Springer, Heidelberg.

[73] E. Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer Berlin Heidelberg, 2008.

[74] G. Wachsmuth. Metamodel adaptation and model co-adaptation. *ECOOP 2007 - Object-Oriented Programming*, 4609:600–624, 2007. Lecture Notes in Computer Science.

Appendix A

Modifications & Refactorings

Table A.1 depicts a list of model modification operations per category. Each of these modification operations are paired with their corresponding database refactorings.

Model Modifications	**Database Refactorings**
Structural Model Modification	
Entity Creation	Introduce New Table
Entity Removal	Drop Table
Attribute Creation	Introduce New Column
Attribute Removal	Drop Column
Relationship Creation	Add Foreign Key Constraint
Relationship Removal	Drop Foreign Key Constraint
Non-Structural Model Modification	
Entity Rename	Rename Table
Attribute Rename	Rename Column
Attribute Change DataType	Introduce New Column & Move Data & Remove Column
Attribute Make PrimaryKey	Introduce Column Constraint
Attribute Drop PrimaryKey [1]	Drop Column Constraint
Super Type Entity Creation	Introduce New Table, New Column, Column Constraint & Add Foreign Key Constraint
Super Type Entity Removal	Drop Foreign Key Constraint & Drop Table
Composite Relationship Creation	Introduce Column Constraints & Add Foreign Key Constraint
Composite Relationship Switch	Add Foreign Key Constraint & Drop Foreign Key Constraint
Complex Model Modification	
Extract Entity	Introduce New Table & New Column & Move Column(s) & Introduce Column Constraint & Add Foreign Key Constraint
Inline Entity	Drop Table & Drop Foreign Key Constraint & Move Column(s)
Fold Entity	Add Foreign Key Constraint & Move Column(s)
Unfold Entity	Drop Foreign Key Constraint & Move Column(s)
Move attribute over rel.	Move Column
Collect attribute over rel.	Move Column
Semantic Model Modification	
Merge Entity	Merge Tables
Split Entity	Split Tables
Merge Attribute	Merge Columns

Table A.1: Model Modification & Corresponding Database Transformation

[1] Drop the constraint, not the attribute itself

Appendix B

Generated Data Models

Figures B.1,B.2,B.3 and B.4 depict the data models used for the evaluation of the proposed operation-based approach redesign. Because of the size of these data models each one will be addressed in separate sections. Furthermore, the purpose of this research project was to address data model evolution, therefore, this chapter will focus on reporting the changes made to the generated models and not the details of the models themself.

The data models related to the account data model (IM_1) are outlined in Section B.1. Sections B.2, B.3 and B.4 describe the contact, lead and opportunity data models, respectively.

B.1 Account Related Data Models

Figure B.1 gives an overview of the structure of this data model and tables B.1-B.4 detail the modifications that were made.

Modification Type	**Model Modifications**
Entity Creation	Twitter, LinkedIn and Facebook
Entity Removal	Twitter, LinkedIn and Facebook
Attribute Creation	each entity received 6 similar attributes (e.g. Twitter: twitterID: int, accountID: int, twitter_username: int, twitter_webpage: int, twitter_postedText: int and twitter_checkdate: int)
Attribute Removal	(all of the newly create attributes)
Relationship Creation	(each entity received a link with the account entity, establishing a FK constraint)
Relationship Removal	(all of the newly created links)

Table B.1: Overview of Structural Model Modifications[1]

[1]These transformations were used to create the target model TM-struct $_1$

Modification Type	Model Modifications
Entity Rename	TwitterProfile, LinkedInProfile, FacebookProfile
Attribute Rename	each entity received similar renames (e.g. Twitter: tw_ID, accountID, tw_username, tw_webpage, tw_postedtext and tw_checkdate)
Attribute Change DataType	each entity received similar changes (e.g. Twitter: tw_ID: int, accountID: uniqueidentifier, tw_username: nvarchar(160), tw_webpage: Text, tw_postedtext: nvarchar(250), tw_checkdate: datetime)
Attribute Make PrimaryKey	the 'ID' attribute for each new entity was made the PK (e.g. Twitter: tw_ID **(pk)**)
Attribute Drop PrimaryKey	(all of the newly created PKs)
Super Type Entity Creation	superType entity: SocialMediaMarketing (socialmediaMrktID (PK), accountID(FK with entity account), subTypes: TwitterMrkt (twitterMrktID (PK, FK with superType), hashtag_campagne), LinkedInMrkt (linkedInMrktID (PK, FK with superType), linkedIn_groupID), FacebookMrkt (facebookMrktID (PK, FK with superType), announcement_campagne)
Super Type Entity Removal	(removal of SocialMediaMarketing)
Composite Relationship Creation	(see relationships in superType creation)
Composite Relationship Switch	(defined as the deletion of current links and recreate the links to another superType entity, thus see the changes related to creation and removal of superType entity above)

Table B.2: Overview of Non-Structural Model Modifications[2]

Modification Type	Model Modifications
Extract Entity	(all address related attributes in 'account' were moved to the newly created 'addressInfo' entity. Additionally, 2 attributes were created: addressInfoID (PK) and accountID(FK with account)
Inline Entity	(all address related attributes were moved back to 'account' and 'addressInfo' was removed
Fold Entity	(see extract mod. without the entity creation)
Unfold Entity	(see inline mod. without the entity removal)
Move attribute over rel.	(see extract mod. without the entity creation)
Collect attribute over rel.	(see inline mod. without the entity removal)

Table B.3: Overview of Complex Model Modifications[3]

Modification Type	Model Modifications
Merge Entity	entity addressInfo was created (see extract mod.) and then 'addressInfo' was merged with 'account'
Split Entity	(this operation was used to execute the extract mod.)
Merge Attribute	the attributes 'address1_city' and 'address1_country' were merge by concatenating there values into 'address1_city_country'

Table B.4: Overview of Semantic Model Modifications[4]

[2]These transformations were used to create the target model TM-nonStruct$_1$

[3]These transformations were used to create the target model TM-complex$_1$

[4]These transformations were used to create the target model TM-semantic$_1$

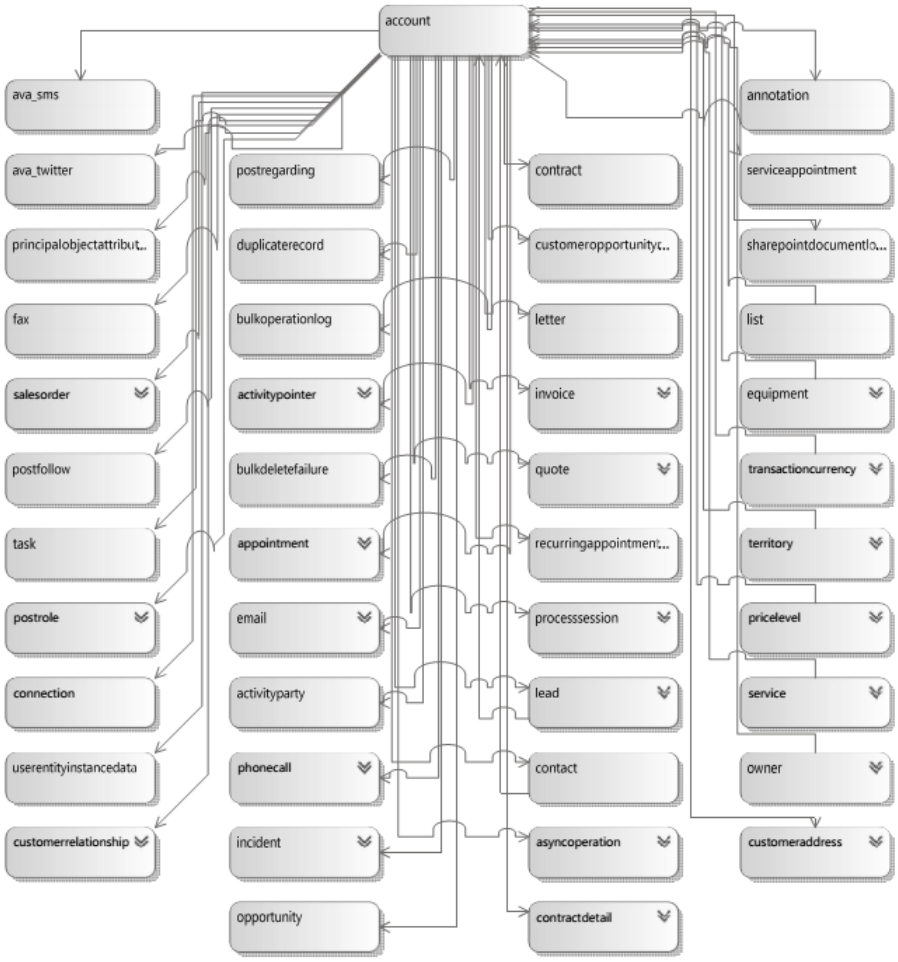

Figure B.1: Initial Data Model based on Account Data (IM_1)

B.2 Contact Related Data Models

Figure B.2 gives an overview of the structure of this data model and table B.5 detail the modifications that were made.

Modification Type	Model Modifications
Extract Entity	(all address related attributes in 'contact' were moved to the newly created 'addressInfo' entity. Additionally, 2 attributes were created: addressInfoID (PK) and contactID(FK with contact)
Inline Entity	(all attributes starting with 'address1_*' were moved back to 'contact'
Fold Entity	(see extract mod. without the entity creation)
Unfold Entity	(see inline mod. without the entity removal)
Move attribute over rel.	(see extract mod. without the entity creation)
Collect attribute over rel.	(see inline mod. without the entity removal)

Table B.5: Overview of Complex Model Modifications

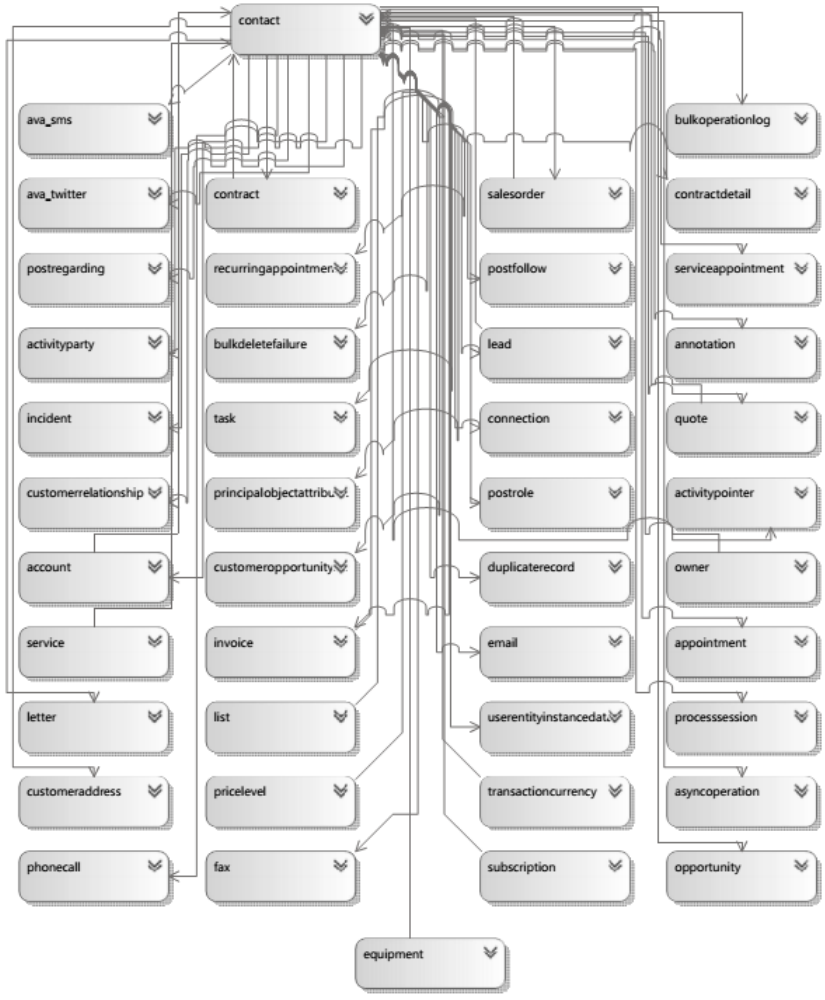

Figure B.2: Initial Data Model based on Contact Data (IM_2)

B.3 Lead Related Data Models

Figure B.3 gives an overview of the structure of this data model and table B.6 detail the modifications that were made.

Modification Type	Model Modifications
Extract Entity	(all dotNot-flag related attributes in 'lead' were moved to the newly created 'DoNotInfo' entity. Additionally, 2 attributes were created: donotInfoID (PK) and leadID(FK with lead))
Inline Entity	(all the moved attributes are moved back and the donotInfo entity was removed)
Fold Entity	(see extract mod. without the entity creation)
Unfold Entity	(see inline mod. without the entity removal)
Move attribute over rel.	(see extract mod. without the entity creation)
Collect attribute over rel.	(see inline mod. without the entity removal)

Table B.6: Overview of Complex Model Modifications

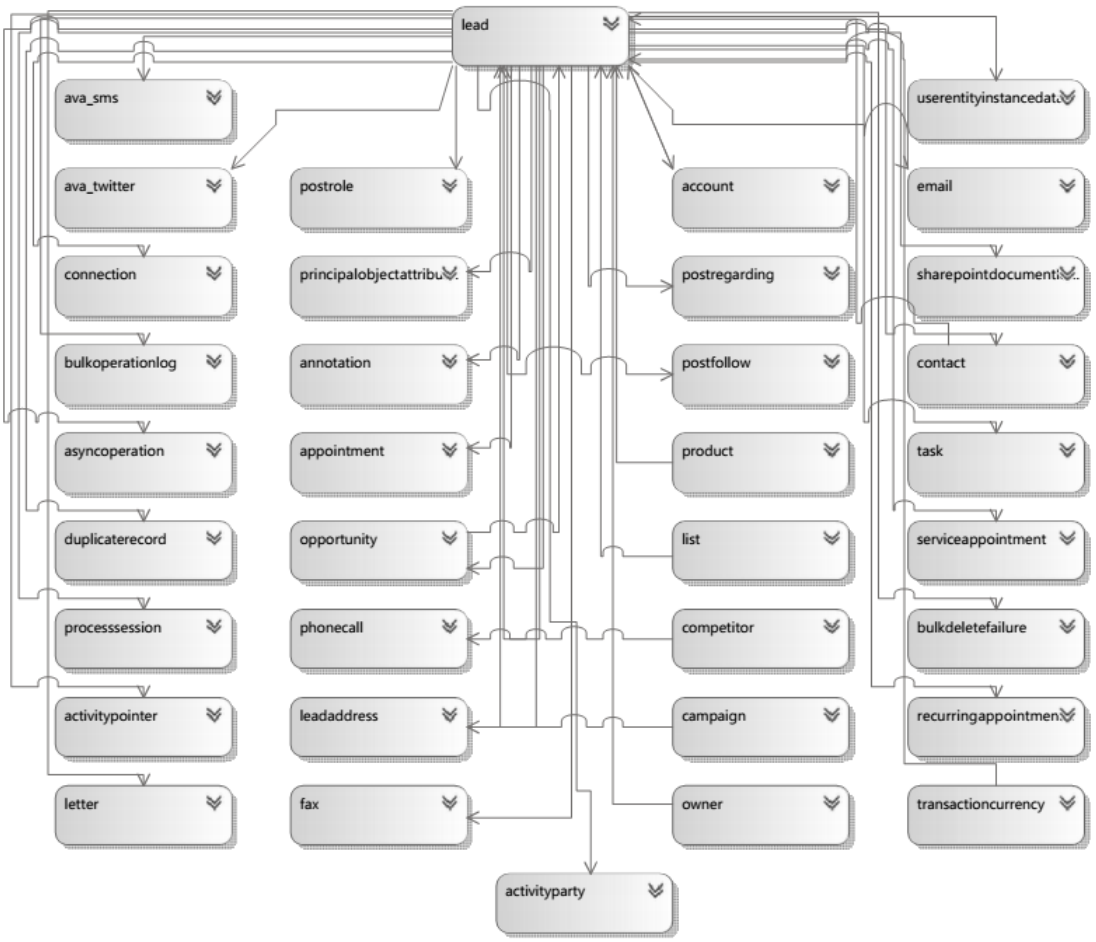

Figure B.3: Initial Data Model based on Lead Data (IM_3)

B.4 Opportunity Related Data Models

Figure B.3 gives an overview of the structure of this data model and table B.7 detail the modifications that were made.

Modification Type	Model Modifications
Extract Entity	(all monetary-total related attributes in 'opportunity' were moved to the newly created 'TotalInfo' entity. Additionally, 2 attributes were created: totalInfoID (PK) and opportunityID(FK with opportunity))
Inline Entity	(all the moved attributes are moved back and the totalInfo entity was removed)
Fold Entity	(see extract mod. without the entity creation)
Unfold Entity	(see inline mod. without the entity removal)
Move attribute over rel.	(see extract mod. without the entity creation)
Collect attribute over rel.	(see inline mod. without the entity removal)

Table B.7: Overview of Complex Model Modifications

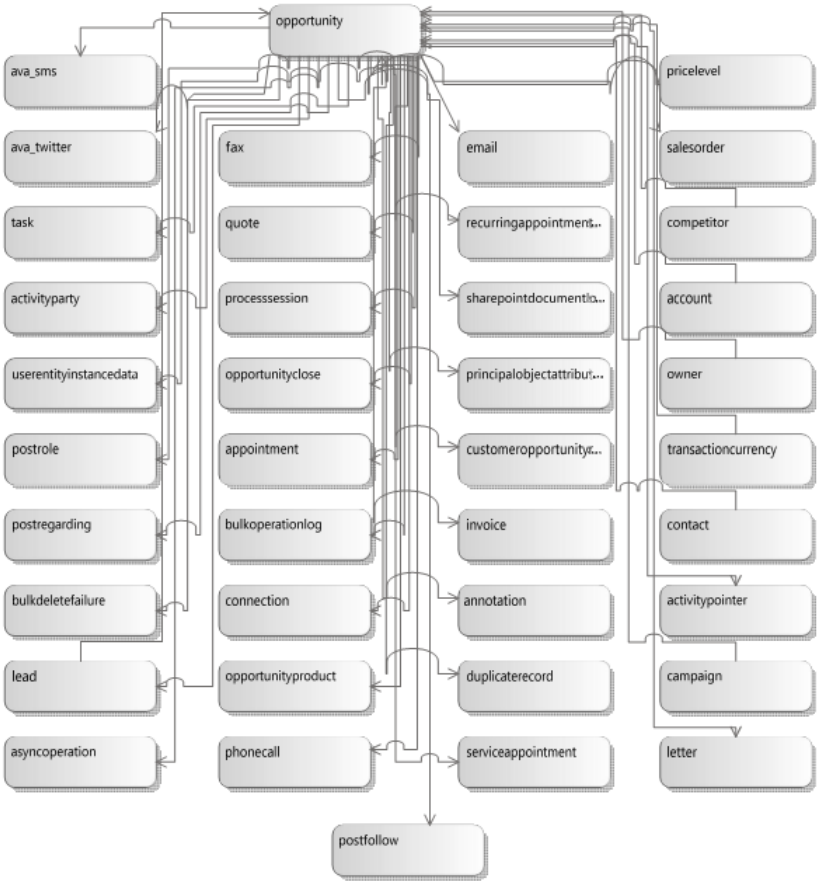

Figure B.4: Initial Data Model based on Opportunity Data (IM_4)

Appendix C

Operator LOC Reduction

The tables in this appendix present the amount of Lines-Of-Code (LOC) reduction that was achieved by replacing the original operator definitions with their composite counter-parts.

Operators	LOC Composable Op.	LOC Composite Op.	LOC Reduction	Reduction Percentage
Entity Creation	8	1	7	87.5%
Entity Removal	5	1	4	80%
Attribute Creation	11	1	10	90.9%
Attribute Removal	5	1	4	80%
Relationship Creation	7	1	6	85.7%
Relationship Removal	5	1	4	80%
Average Reduction	-	-	-	84%

Table C.1: Implementation Reduction for Structural Modifications

Operators	LOC Composable Op.	LOC Composite Op.	LOC Reduction	Reduction Percentage
Entity Rename	10	1	9	90%
Attribute Rename	10	1	9	90%
Attribute Change DataType	5	1	4	80%
Attribute Make PrimaryKey	8	1	7	87.5%
Attribute Drop PrimaryKey	8	1	7	87.5%
Super Type Entity Creation	16	5	11	68.8%
Super Type Entity Removal	7	5	2	28.6%
Composite Relationship Creation	8	1	7	87.5%
Composite Relationship Switch	9	1	8	88.9%
Average Reduction	-	-	-	78.8%

Table C.2: Implementation Reduction for Non-Structural Modifications

Operators	LOC Composable Op.	LOC Composite Op.	LOC Reduction	Reduction Percentage
Extract Entity	23	4	19	82.6%
Inline Entity	15	1	14	93.3%
Fold Entity	10	1	9	90%
Unfold Entity	15	1	14	93.3%
Move attribute over rel.	7	1	6	85.7%
Collect attribute over rel.	7	1	6	85.7%
Average Reduction	-	-	-	88.4%

Table C.3: Implementation Reduction for Complex Modifications

Operators	LOC Composable Op.	LOC Composite Op.	LOC Reduction	Reduction Percentage
Merge Entity	22	8	14	63.6%
Split Entity	17	7	10	85.8%
Merge Attribute	14	1	13	92.9%
Average Reduction	-	-	-	80.8%

Table C.4: Implementation Reduction for Semantic Modifications

Appendix D

SQL Server Data Type Conversion

The data type conversions that are allowed for SQL Server system-supplied data types can be divided into three categories: those that do not require any extra effort (i.e. implicit conversion), those that require a 'cast' or 'convert' function (i.e. explicit conversion), and conversions that are not allowed. Figure D.1 illustrates all implicit and explicit conversions that are allowed [1].

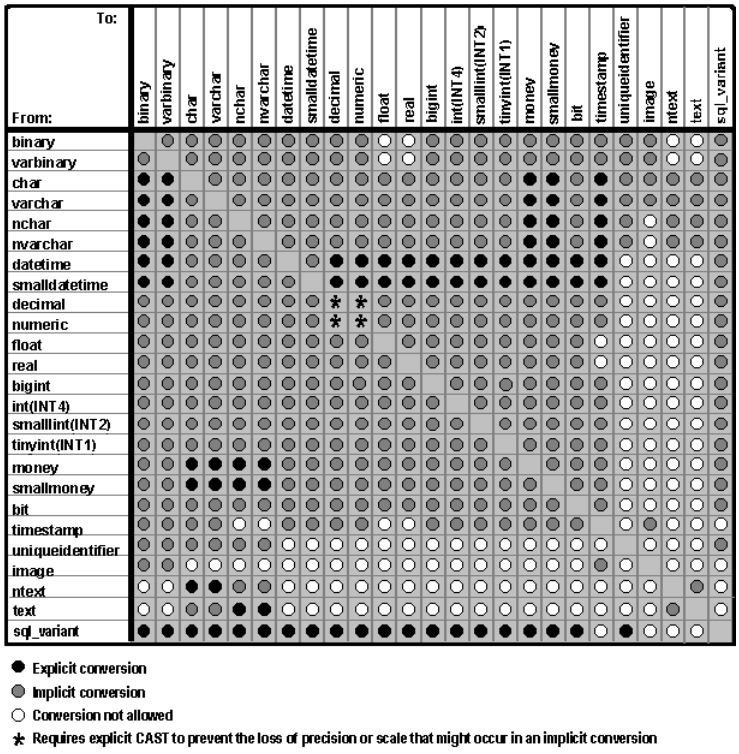

Figure D.1: SQL Server Data Type Conversion Chart

[1] Sourced from: `http://www.microsoft.com/en-us/download/details.aspx?id=35834`

Listing D.1 contains the partial code used to map data type updates. It shows how the data conversion is implemented. The code has been truncated for the sake of clarity.

```
1  string getSQLStringForDataTypeConversion(entity, attribute,
       oldType, newType)
2  {
3    ...
4    // Implicit conversion of data type
5    if (hasImplicitConversion)
6    {
7      strbSqlCode.AppendFormat(@"ALTER TABLE [{0}].{1}" +
           Environment.NewLine, SHEMA_NAME, entityName);
8      strbSqlCode.AppendFormat(@"    ALTER COLUMN {0} {1}" +
           Environment.NewLine, attribute.CurrentName,
           MappingTypeConvertion.translateDataTypeIntoSQLString(
           newType));
9      strbSqlCode.AppendLine("GO");
10   }
11   else if (hasExplicitConversion)
12   {
13     // Copy data to temp column, while converting it to the proper
           type (ETL)
14     strbSqlCode.AppendFormat(@"ALTER TABLE [{0}].{1} ADD {2} {3}"
           + Environment.NewLine, SHEMA_NAME, entityName, strPrefix +
            attribute.CurrentName, MappingTypeConvertion.
           translateDataTypeIntoSQLString(newType));
15     strbSqlCode.AppendFormat(@"UPDATE [{0}].{1} SET" + Environment
           .NewLine, SHEMA_NAME, entityName);
16     //set new_column_name = convert(decimal(18,2), old_column_name
           )
17     strbSqlCode.AppendFormat(@"    {0} = cast({1} as {2})" +
           Environment.NewLine, strPrefix + attribute.CurrentName,
           attribute.CurrentName, MappingTypeConvertion.
           translateDataTypeIntoSQLString(newType));
18     strbSqlCode.AppendLine("GO" + Environment.NewLine);
19
20     strbSqlCode.AppendFormat("ALTER TABLE [{0}].{1}" + Environment
           .NewLine, SHEMA_NAME, entityName);
21     strbSqlCode.AppendFormat("    DROP COLUMN {0}" + Environment.
           NewLine, attribute.CurrentName);
22     strbSqlCode.AppendLine("GO" + Environment.NewLine);
23
24     strbSqlCode.AppendFormat("EXEC sp_RENAME '{0}.{1}.{2}', '{3}',
            'COLUMN' " + Environment.NewLine, SHEMA_NAME, entityName,
            strPrefix + attribute.CurrentName, attribute.CurrentName)
           ;
25     strbSqlCode.AppendLine("GO");
26   }
27   return strbSqlCode.ToString();
28 }
```

Listing D.1: Mapping Implementation - Data Type Conversion

Appendix E

Database Transformation Adjustments

This appendix contains two listings (listing E.1 and E.2) which show how an incorrect database transformation is adjusted to correctly perform a database transformation for the extract operation mentioned in Section 5.2.3. The code inside the listings are truncated for the sake of clarity.

```
1  /*** --- Creating table for added entity 'AddressInfo' --- ***/
2  CREATE TABLE [dbo].AddressInfo (
3      addressInfo_ID  INT,
4      account_ID UNIQUEIDENTIFIER
5  );
6  GO
7
8  /*** --- Updating key contraint for table AddressInfo on column
       addressInfo_ID --- ***/
9  ALTER TABLE [dbo].AddressInfo
10     ADD CONSTRAINT PK_addressInfo_ID
11         PRIMARY KEY (addressInfo_ID);
12 GO
13
14 /*** --- Updating key contraint for table AddressInfo on column
       account_ID --- ***/
15 /*** --- Attribute is now foreign key --- ***/
16 ALTER TABLE [dbo].AddressInfo
17     ADD CONSTRAINT FK_AddressInfo_account_ID
18         FOREIGN KEY (account_ID)
19         REFERENCES account;
20 GO
21
22 /*** --- Moving column statecode from source table account to
       target table AddressInfo --- ***/
23 ALTER TABLE [dbo].AddressInfo ADD statecode  VARCHAR (255)
24 GO
25
26 /*** --- Moving column address1_addressid from source table
       account to target table AddressInfo --- ***/
27 ALTER TABLE [dbo].AddressInfo ADD address1_addressid
       UNIQUEIDENTIFIER
28 GO
29
30 INSERT INTO [dbo].AddressInfo (address1_addressid)
31     SELECT address1_addressid
32     FROM [dbo].account
33 GO
34
35 /*** --- Moving column address1_addresstypecode from source table
       account to target table AddressInfo --- ***/
36 ALTER TABLE [dbo].AddressInfo ADD address1_addresstypecode
       VARCHAR (255)
37 GO
38
39 INSERT INTO [dbo].AddressInfo (address1_addresstypecode)
40     SELECT address1_addresstypecode
41     FROM [dbo].account
42 GO
43 ...
```

Listing E.1: Extract Entity - Generated Database Transformation

```
CREATE TABLE [dbo].[AddressInfo](
  [addressInfo_ID] [int] IDENTITY(1,1) NOT NULL,
  [account_ID] [uniqueidentifier] NOT NULL,
 CONSTRAINT [PK_AddressInfo] PRIMARY KEY CLUSTERED
(
  [addressInfo_ID] ASC
)
GO

/*** --- Establish relationship between the tables --- ***/
ALTER TABLE [dbo].[AddressInfo] ADD CONSTRAINT [
    FK_AddressInfo_account] FOREIGN KEY([account_ID])
REFERENCES [dbo].[account] ([AccountId])
GO

INSERT INTO [dbo].AddressInfo (account_ID)
  SELECT DISTINCT accountID FROM [dbo].account
GO

/*** --- Moving column address1_addressid from source table
    account to target table AddressInfo --- ***/
ALTER TABLE [dbo].AddressInfo ADD address1_addressid
    UNIQUEIDENTIFIER
GO

UPDATE [dbo].AddressInfo SET address1_addressid =
    (SELECT address1_addressid FROM account
    WHERE accountid = [AddressInfo].account_ID
    )
GO

/*** --- Moving column address1_addresstypecode from source table
    account to target table AddressInfo --- ***/
ALTER TABLE [dbo].AddressInfo ADD address1_addresstypecode
    NVARCHAR (255)
GO

UPDATE [dbo].AddressInfo SET address1_addresstypecode =
    (SELECT address1_addresstypecode FROM account
    WHERE accountid = [AddressInfo].account_ID
    )
GO
...
```

Listing E.2: Extract Entity - Adjusted Database Transformation