UNIVERSIDAD DE MURCIA

DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS

An Approach for Model-Driven Data Reengineering

Un Enfoque de Reingeniería de Datos Dirigido por Modelos

D. Francisco Javier Bermúdez Ruiz

2016

An Approach for Model-Driven Data Reengineering

A dissertation presented by
Francisco Javier Bermúdez Ruiz
and supervised by
Jesús Joaquín García Molina & Oscar Díaz García

In partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the subject of Computer Science

University of Murcia
January 2016

Un Enfoque de Reingeniería de Datos Dirigido por Modelos

Resumen extendido de la tesis

La Ingeniería de Datos es la disciplina de Informática que se encarga de los principios, técnicas, métodos y herramientas para permitir la gestión de los datos en el desarrollo de software. Los datos de una aplicación son habitualmente almacenados en sistemas gestores de bases de datos (por ejemplo, bases de datos relacionales, orientadas a objetos o NoSQL). La Ingeniería de Datos se ha centrado principalmente en los datos relacionales, aunque actualmente está aumentando el interés hacia aproximaciones no relacionales, como las bases de datos NoSQL. En esta tesis, se han abordado cuestiones relacionadas con algunos de los principales temas de la Ingeniería de Datos como: reingeniería de datos, ingeniería inversa de datos, integración de datos entre distintas herramientas y herramientas para la gestión de procesos de reingeniería de datos. Más concretamente en esta tesis se ha explorado la aplicación de técnicas de Ingeniería Software Dirigidas por Modelos (Model-Driven Software Engineering, MDSE o simplemente MDE) en las cuestiones de Ingeniería de Datos mencionadas anteriormente.

MDE pone énfasis en el uso sistemático de modelos para la mejora de la productividad en el desarrollo de software y en algunos aspectos sobre la calidad del software tales como el mantenimiento o la interoperabilidad. Las técnicas MDE han probado ser útiles no solo en el desarrollo de nuevas aplicaciones software, sino también en la reingeniería de sistemas legados. Modelos y metamodelos proporcionan un alto grado de formalismo con el que representar los artefactos comúnmente manipulados en las diferentes etapas de un proceso de evolución de software (por ejemplo, una migración de aplicaciones) mientras que las transformaciones de modelos permiten la automatización de las tareas de evolución a ser aplicadas. Aunque recientemente se han presentado algunas propuestas y experiencias de reingeniería de software dirigida por modelos, la mayoría se han centrado principalmente en el código de las aplicaciones mientras que las cuestiones relativas a la reingeniería de datos han sido pasadas por alto.

Un proceso de reingeniería de datos debe ser cubierto a través de tres dimensiones: conversión de esquema, conversión de datos y conversión de programas. Esta tesis está centrada en la primera dimensión de la reingeniería de datos. En concreto, se presenta un proceso basado en MDE para la conversión de esquemas cuyo propósito es mejorar la calidad del esquema lógico dentro de un escenario de migración de una base de datos relacional. El enfoque propuesto se organiza siguiendo las tres etapas de un proceso de reingeniería: primero se aplica **ingeniería inversa** para extraer una descripción lógica del sistema; a continuación se realiza la **reestructuración** del sistema a través de transformaciones de las descripciones lógicas obtenidas en la etapa anterior; finalmente se ejecuta la etapa de **ingeniería directa** o generación del nuevo sistema a partir de las nuevas descripciones lógicas. Cada etapa has sido implementada mediante cadenas de transformación de modelos que incluyen transformaciones de tres tipos: texto a modelo (T2M), modelo a modelo (M2M) y modelo a texto (M2T). Nuestro proceso no solo realiza una migración tecnológica si no que también incluye una conversión de esquema de base de datos con el que proporcionar mejoras en la calidad de los datos.

La mayoría de los sistemas gestores de bases de datos relacionales actuales disponen de un control para la aplicación de restricciones de integridad referencial (implementado mediante claves ajenas). Sin embargo, son muchos los sistemas de información legados que no hacen uso de este importante control, principalmente si el diseño de la base de datos precedió a una posterior disponibilidad del soporte para las claves ajenas. La detección de claves ajenas implícitas en sistemas legados ha sido un tema de investigación desde hace tiempo en la comunidad de reingeniería de base de datos. Se han propuesto una gran variedad de métodos diferentes, los cuales están basados en el análisis de los tres tipos de artefactos que conforman un sistema de información de datos: esquema, código de la aplicación y datos. El proceso de conversión de esquemas aquí propuesto mejora la calidad de los datos de un sistema mediante la elicitación de claves ajenas a través de estrategias basadas en el análisis del esquema, de los datos y de código de los programas.

La primera etapa de nuestro proceso comienza inyectando los datos del sistema legado en modelos. Para ello se han usado como artefactos fuente los scripts SQL de definición del esquema y los datos de la base de datos, así como el código fuente de la aplicaciones legadas que se encarga del acceso a los datos. A continuación, sobre los modelos extraídos en la primera etapa se aplica un proceso de inferencia o elicitación de claves ajenas implícitas en los datos (aunque no definidas sobre el propio esquema). Los tres análisis anteriormente mencionados han sido implementados combinando diferentes estrategias basadas todas en el uso de técnicas MDE. Ya en la segunda etapa del proceso, mediante un asistente se solicita la intervención manual de los desarrolladores de la migración para confirmar el descubrimiento individual de cada una de las claves ajenas propuestas por nuestra solución. Una vez validada la elicitación, se añaden las nuevas claves ajenas al modelo que representa el esquema y los datos de la base de datos legada. Se aplica entonces un proceso de normalización que se encarga de comprobar la forma normal de cada relación del esquema y se aplican los algoritmos de normalización (descomposición y/o síntesis), si fuera necesario, para dejar el esquema en al menos tercera forma normal. En este etapa, además de intervenir diversas transformaciones de modelos, es necesario integrar el uso de la herramienta Concept Explorer (ConExp) que nos permite la identificación automática de dependencias funcionales necesarias para la normalización. En la tercera y última etapa del proceso se generan los artefactos del nuevo sistema software: los nuevos scripts SQL de base de datos y/o el código JPA (JAva Persistence API) para el acceso y generación del nuevo esquema. El enfoque aquí propuesto se ha validado a través de su aplicación en una base de datos real de un sistema legado, en concreto OSCAR (Open Source Clinical Application Resource), una herramienta para el cuidado de la salud que tiene un amplio uso fundamentalmente en centros hospitalarios de Canadá desde hace mas de 15 años. Se ha proporcionado también una valoración del uso de las técnicas basadas en modelos en nuestra implementación.

Debido a que la evidencia empírica en la elicitación de claves ajenas en sistemas industriales a gran escala es escasa y a menudo los problemas (casos de estudio) usados son cuidadosamente seleccionados para cumplir con una solución particular, en lugar de que fuera al revés, en este trabajo también hemos llevado a cabo un enfoque distinto: aplicar una reingeniería sobre un sistema de información real y crítico. Las dimensiones y complejidad del caso de estudio han conllevado realizar un nuevo análisis complementario (en este caso manual) que ha consistido en la triangulación de los resultados obtenidos en los anteriores análisis. Se han definido

varios criterios para la aceptación de claves ajenas candidatas que hayan sido descubiertas en el esquema, en los datos o en el código de la aplicación, y se ha proporcionado una discusión acerca de la confiabilidad de los resultados finales.

Tal y como se ha comentado, el proceso de conversión de esquemas también ha considerado la comprobación y corrección automática (si fuera necesario) del nivel de normalización de la base de datos, para lo que ha sido necesario integrar la herramienta ConExp en nuestro proceso de reingeniería de datos MDE. Con esta experiencia, y a partir del conocimiento obtenido de la integración, hemos continuado explorando las capacidades que ofrece MDE en al área de la interoperabilidad de herramientas mediante la creación de un puente *MDI* (Model-Driven Interoperability) entre nuestro proceso y la herramienta de ingeniería de datos DB-Main. Un puente bidireccional entre DB-Main y la tecnología MDE (también conocida como Modelware) está formado por un metamodelo que representa la información manejada por la herramienta (denominado metamodelo pivote) junto con un *inyector* que genera modelos a partir de la información proporcionada por la herramienta y un *extractor* que realiza la operación inversa. Estos dos últimos artefactos conforman lo que se conoce en MDI como correspondencia sintáctica de un puente de interoperabilidad. Puesto que DB-Main ofrece diferentes alternativas para acceder a sus datos (un API de métodos Java, un fichero de exportación de datos en formato XML y un fichero en formato propietario con todos los datos de un proyecto), hemos decidido evaluar diferentes estrategias para implementar la correspondencia sintáctica (es decir, la creación de extractores e inyectores de modelos). Una vez completado este puente, se ha realizado la construcción de un último puente bidireccional que, en este caso, requiere la implementación de una correspondencia semántica entre los datos de dos herramientas a integrar representados a través de los correspondientes metamodelos pivote. Para ejemplificar este puente se tomó como herramientas la antes mencionada DB-Main y la herramienta de ingeniería de requisitos Objectiver. Esto nos ha permitido experimentar con la aplicación del lenguaje QVT-Relational para implementar una correspondencia semántica bidireccional dentro del puente de interoperabilidad.

Uno de los principales retos en la adopción de MDE en procesos de construcción de software de gran escala y complejos es todavía la disponibilidad de herramientas para el soporte de procesos basados en técnicas MDE y que permitan integrar la ejecución de tareas manuales y automáticas. Por lo tanto, en esta tesis también se ha propuesto una herramienta con la que proporcionar el soporte para la definición y la ejecución de procesos de migración MDE en general, y que soporta la ejecución de nuestro proceso de reingeniería de datos en particular. La ausencia de entornos de desarrollo capaces de integrar la ejecución de tareas automáticas junto con la de tareas manuales que sean aplicadas por los desarrolladores de la migración ha motivado la creación de la herramienta descrita en esta tesis. La herramienta tiene como objetivos permitir: la definición de planes de migración, la instanciación de esos planes sobre proyectos de migración reales (es decir, sistemas de información legados) y la ejecución (mediante un intérprete) de los planes de migración instanciados sobre sistemas legados. Para ello, primero se ha abordado la creación de un lenguaje para la definición de modelos que representen planes de migración. Se trata de un lenguaje específico de dominio (DSL) basado en SPEM y orientado a la definición de procesos de migración implementados mediante tareas MDE. Para cada proyecto de migración particular, estos modelos son instanciados con el objetivo de contener toda la información necesaria para la ejecución real del proceso (por ejemplo, las rutas de los recursos y las herramientas de transformación). Una vez instanciado un plan de migración para un proyecto, los modelos son ejecutados mediante un interprete de procesos encargado de generar tickets en un servidor Trac que se corresponden con: scripts de tareas Ant para la ejecución automática y entradas en Mylyn para que la gestión manual de tareas pueda ser integrada en un entorno de desarrollo. Puesto que tanto las tareas manuales como las automáticas son generadas a partir de tickets en Trac, este tipo de servidor permite definir dependencias entre tareas que derivan en bloqueos que garantizan que una tarea no se ejecute hasta que todas sus dependencias han sido resueltas (tareas completadas). Asimismo, el uso de una herramienta como Mylyn permite la asignación de tareas manuales a los desarrolladores de la migración, facilitando por tanto la gestión de los recursos en los proyectos de migración así como el seguimiento y control del proyecto por parte del responsable del mismo.

Resumiendo, el objetivo de esta tesis ha sido aplicar técnicas MDE en un escenario de de reingeniería de datos con tres propósitos diferentes, pero relacionados: abordar la conversión de esquemas durante un proceso de reingeniería de datos común en proyectos de migración, proporcionar la implementación de un enfoque

que nos permita la integración de una herramienta de reingeniería de datos con otras herramientas software y construir una herramienta capaz de automatizar la definición y ejecución de procesos de migración. Con los dos primeros objetivos, se han investigado los beneficios de usar una implementación MDE con respecto a abordarla con técnicas tradicionales. Con el último objetivo se ha evaluado cómo las técnicas MDE pueden ser útiles para desarrollar herramientas para el soporte de procesos software.

An Approach for Model-Driven Data Reengineering

Abstract

Data Engineering is the Computer Science discipline concerned with the principles, techniques, methods and tools to support the data management in the software development. Data are normally stored in database management systems (e.g. Relational, Object-oriented or NoSQL) and Data Engineering has been mainly focused on relational data so far, although interest is shifting towards NoSQL databases. In this thesis, we have addressed issues which are related to some of the main topics of Data Engineering, such as Data Reengineering, Data Reverse Engineering, Data Integration and Data Tooling. More specifically, we have explored the application of Model-Driven Engineering (MDE) in data engineering.

MDE emphasizes the systematic use of models to improve software productivity and some aspects of the software quality, such as maintainability or interoperability. Model-driven techniques have proven useful not only as regards developing new software applications but also the reengineering of legacy systems. Models and metamodels provide a high-level formalism with which to represent artefacts commonly manipulated in the different stages of a software evolution process (e.g., a software migration) while model transformation allows the automation of the evolution tasks to be performed. Some approaches and experiences of model-driven software reengineering have recently been presented but they have been focused on the code while data reengineering aspects have been overlooked.

A data reengineering should be covered through three dimensions: schema conversion, data conversion and program conversion. This thesis focuses on the first dimension of data reengineering. We present *an MDE-based process for the schema conversion* whose purpose is to improve the quality of the logical schema in a relational database migration scenario. The approach proposed is organised following the three stages of a software reengineering process: reverse engineering, restructuring and forward engineering. Each stage has been implemented by means of

model transformation chains. We have validated our approach through its application to a real widely-used database. We also provide an assessment of the use of MDE techniques in our implementation.

Our process applies not only a technological migration but also a schema conversion with which to provide data quality improvements. Most modern relational database management systems have the ability to monitor and enforce referential integrity constraints (implemented by foreign key) but heavily evolved legacy information systems may not make use of this important feature, if their design predates its availability.

The *detection of implicit foreign keys* in legacy systems has been a long-term research topic in the database reengineering community and a variety of different methods have been proposed, which are based on the analysis of the three kinds of artefacts that form a data-intensive information system, namely schema, application code and data. Our schema conversion process improves the data quality of a system by eliciting foreign keys through strategies of analysis of schema, data and application code analysis.

Owing to empirical evidence on eliciting foreign keys in large-scale industrial systems is scarce and often ”problems” (case studies) are carefully selected to fit a particular ”solution” (method), rather than the other way around, we also carry out a different approach: we re-engineer a real, complex and mission-critical information system and it leads to a new manual and complementary analysis that consists of *the triangulation of the results* obtained in all of our previous analysis. We define several criteria for the acceptance of the candidate foreign keys discovered in the schema, data and application code analysis and we discuss the final results.

The schema conversion process has also considered checking and fixing automatically, if necessary, the database normalisation level. For this task, we have integrated the Concept Explorer tool (ConExp) into our MDE solution in order to identify functional dependencies in a relational database. From the knowledge gained from the integration of ConExp, we explore the MDE capabilities for the tool interoperability through the creation of a bidirectional bridge between the DB-Main data reengineering tool and the Objectiver requirement engineering tool. As DB-Main offers several alternatives to access its data (API, and XML

and proprietary formats), we have evaluated different strategies to implement the syntactic mapping (i.e. creating model injectors and extractors). In addition, we have explored the use of QVT relational to implement the semantic mapping.

One of the main challenges in the adoption of MDE in large and complex processes is still the availability of tools to support MDE-based processes which integrate manual and automated tasks. Therefore, in this thesis we have also proposed *a tool with which to provide the definition and enactment of MDE migration processes* in general, and which supports *the execution of our data reengineering process* in particular. The lack of IDEs capable of integrating the execution of automated tasks and manual tasks to be performed by developers has motivated the creation of the tool described in this thesis. We have defined a SPEM-based language for defining models that represent migration plans. For each particular migration project, these models are instantiated in order to contain all the information needed for the enactment (e.g. resource paths and transformation tools). Then, these models are enacted by means of a process interpreter which generates Ant scripts to execute the automated tasks and Trac tickets for managing manual tasks with the Mylyn tool.

Summarizing, the aim of our work has been applying MDE techniques with three different purposes: tackling the schema conversion during a data reengineering, approaching the integration of a database reengineering tool with other software tools, and building a tool able to automate the development of migration processes. With the two former, we have investigated the benefits of MDE with regard to traditional solutions, and the latter objective addressed how MDE may be useful to develop tools supporting software processes.

Contents

Listing of figures

List of Tables

Agradecimientos

Han sido varios años los invertidos en el trabajo que se recoge ahora en este documento. Multitud de experiencias han ido acompañando a esta tesis, moldeándola con mayor o menor influencia. Y llegó el momento de recordar y agradecer a todas las personas que me han ayudado desde una perspectiva profesional, y por supuesto personal, en el largo recorrido que ha supuesto la realización de la tesis. Pero es justo ahora cuando más dubitativo me encuentro pues intento analizar desde un punto de vista científico cuál sería la forma más correcta de ordenar a los a continuación mencionados. ¿Los incluyo midiendo el impacto que han tenido en mis investigaciones o comunicaciones?... o tal vez ¿debería cuantificar como me han apoyado en lo personal para soportar los baches profesionales en los que he sucumbido en esta carrera hacia el reconocimiento científico? Es una difícil decisión... O quizás no, y es más sencillo que todo lo anteriormente mencionado. Tal vez tan solo deba confiar en la justicia de mi corazón y reconocer que todos los aquí mencionados son parte de la tesis que por fin hoy acabo. Todos por igual, sin importar el orden o si tan solo compartieron conmigo una pequeña fracción de tiempo a lo largo de todos estos años como doctorando. Quiero, por tanto, dar mi agradecimiento a **todos** los aquí presentes. Y puede que algunos puedan pensar que apenas han contribuido al resultado final aquí presentado, pues no fueron muchas las veces que me preguntaron o animaron en mis tareas de investigación. Sin embargo, todos ellos contribuyeron con tan solo traer algo de luz a aquellos oscuros días de trabajo en los sucumbí presa de la frustación.

Voy a empezar dando mi mas profundo agradecimiento a mi familia. A los que me han tenido que soportar todos los días, y han sobrellevado de la mejor manera posible los problema que del trabajo me llevaba muy a menudo. A Claudia y a Raquel, por que son la razón de mi existencia y el motor que me impulsa a sobrepasar los obstáculos de la vida. A mi esposa Antonia Mari, el incondicional apoyo sin el cuál me resultaría imposible imaginar la consecución de mis éxitos. A mis padres, Antonio y Paquita, que germinaron en mi los valores personales y profesionales que tanto me enorgullece demostrar y a quienes anhelo siquiera asemejarme algún día. A mis hermanos y cuñados, Maribel y Jose Manuel, Jose Antonio y Mari Carmen. Y por supuesto a mis queridas sobrinas: Isabel y la recién llegada Gema. Todos ellos me han acompañado en esta travesía y apoyado cuando lo he necesitado.

Quiero por supuesto dar mi agradecimiento a mis directores de tesis, Oscar Díaz y especialmente a Jesús García, pues son los mayores responsables en el ámbito profesional de conseguir hacerme llegar finalmente a la meta. Un carrera larga en la que he invertido más tiempo del esperado, pero en la que por fin hoy rompo la cinta de llegada gracias a que su constancia no me dejó desfallecer en ningún momento y a que me ofrecieron más tiempo del que disponían.

No voy a dejar sin mencionar a mis amigos, Antonio y Verónica, Ricardo y Laura, y por último pero no menos importante, a Fernando. Han supuesto el combustible que necesitaba en mis días fuera de la oficina, y sin el cual hubiera resultado imposible atravesar el largo puente del lunes al viernes. De todos ellos aprendo cada día algo valioso y que seguro que habré reflejado de alguna modo en mi investigación.

Numerosos agradecimientos deben ir a parar a otros amigos, los que he conocido en el laboratorio donde he ido realizando la tesis. Empezaré mencionando de manera especial a Juan Manuel y a Oscar Sánchez. Tal vez los que hayan dejado una huella mas palpable en mis logros investigadores. Muchas son las horas que hemos compartido en nuestro trabajo. Muchas las enseñanzas, conversaciones, cafés, y en general, experiencias. Momentos que estarán para siempre en mi recuerdo. Quiero mencionar también a los amigos que siempre han estado allí donde los necesitaba, que se han preocupado y han respondido con paciencia a todas mis dudas de investigador novato: Jesús Sánchez y Javier Cánovas. No me quiero dejar a Javier Espinazo quien también ha tenido una gran influencia en los resultados que recoge esta tesis. Y a Fernando, Joaquín y Fran, a los que conocí en el laboratorio y con los que fue fácil establecer amistad por la excelencia de sus

personas. Y sería injusto no mencionar a Mari Luz por su apoyo en mi segunda andadura universitaria.

Quiero también dedicar un largo y sentido párrafo a todos aquellos con los que he tenido la poca fortuna de tan solo coincidir un breve tiempo de mi etapa investigadora. Empezaré por aquellos que me ayudaron en una parte importante de la consecución de la tesis: Anthony y Jens. A menudo pienso que realmente me ofrecieron más oportunidades profesionales de las que yo jamás creí merecer. Y si me muevo a tierras belgas, me niego a cerrar mis agradecimientos sin resaltar a los amigos que allí dejé. Mis compañeros de laboratorio Loup y Nesrine, en los que espero haber sembrado la amistad que ellos plantaron en mí. Me resulta imposible olvidarme de los que hicieron que mi estancia en Namur se convirtiera en inolvidable y seguro que irrepetible: Francesco, Silvia, Frédéric y Marco. El buen sabor que dejaron en mi fue comparable al de las deliciosas piadinas de Giuseppe. Quisiera expresar un especial agradecimiento a Sofia, Manal, Saad y Lamya. Su generosa amistad y los días vividos en Marruecos perdurarán para siempre en mi memoria. Ellos me enseñaron el auténtico sentido de la hospitalidad. También quiero dar mi agradecimiento a Paul por su incansable disponibilidad.

Por último, no me gustaría dejar de mencionar a Begoña, Marcos, Joaquín, Rafa, Fabian, James y Jose Ramón.

A todos ellos, muchas gracias.

"Colour my life with the chaos of trouble" from The Boy With The Arab Strap, Belle and Sebastian

"Por el día nos encierran en sus jaulas de cemento y aprendemos del león. Por las noches atrapamos corazones asfixiados y disparos en su honor. Mírame, soy feliz, tu juego me ha dejado así. Consumir, producir, la sangre cubre mi nariz. No sé dónde quedó el rumor que nos vió nacer... pagó la jaula al domador."

from Un día en el mundo, Vetusta Morla
(Suggested by Claudia and Raquel)

"Failure teaches us that life is but a draft, a long rehearsal for a show that will never play."

from Le fabuleux destin d'Amélie Poulain
(Suggested by Antonia Mari)

1 Introduction

Data Engineering is the Computer Science discipline concerned with the principles, techniques, methods and tools to support the data management in the software development. Data are normally stored in database management systems (e.g. Relational, Object-oriented or NoSQL) and Data Engineering has been mainly focused on relational data so far, although interest is shifting towards NoSQL databases. In this thesis, we have addressed issues which are related to some of the main topics of Data Engineering, such as Data Evolution (e.g. data reengineering), Data Reverse Engineering, Data Integration and Data Tooling.

Businesses have legacy applications that were built in the past and have been working since then. However, in the long term, the maintenance of these applications is rather complicated [1]. On the one hand, technologies used or design choices may require a data or software quality improvement. For instance, in the case of data-intensive systems, some legacy databases were defined without considering referential integrity constraints, so they become even more difficult to maintain as time goes by. On the other hand, nowadays there are plenty of technologies and devices that are either more attractive to users or better fit the needs of businesses. All these reasons have meant that a large number of businesses are *modernising* their legacy systems to new platforms (typically web platforms), which better meet their needs of extensibility, maintainability, as well as improving other aspects such as extensibility, performance or distribution.

Application modernisation can be defined as the refactoring, re-purposing or consolidation of legacy software systems in order to align them more closely with current business needs. Reengineering is a kind of software modernisation in which the system quality is improved by means of a systematic process over three stages [2]: reverse engineering, restructuring and forward engineering. Firstly, a reverse engineering stage analyses the existing system and extracts knowledge which is represented at different abstraction levels. A second stage restructures these abstract representations to establish a mapping between the existing system and the target system. Finally, a forward engineering stage obtains the target artefacts from the output of the restructuring stage. A *migration* is a special case of reengineering in which a software system is moved from one technology to another.

Data-intensive systems include two main components: a set of software programs and a database. Data-intensive system reengineering therefore surpasses code so as to also include data. In fact, software system reengineering is traditionally organised into two separated areas: software reengineering and data reengineering. As noted in [3] this separation is particularly clear in reverse engineering in which there have over the years been significant contributions in each field. Software and data reengineering have developed theories, methods and tools which have been adopted by the industry although both communities ”must still face numerous challenges to properly address data-intensive systems” [4].

A mature, stable and functionally complete tooling is one of the most important requirements in the software industry in order to adopt a systematic software process which produces results in accordance with high quality standards. A migration process can benefit tools that automate a specific task of data reengineering or other tools that can simply be useful in completing a data modernisation. For instance, a software process engineering tooling supporting the definition and enactment of migration process could provide a high level of productivity when a application modernisation must be performed.

Over the last few years, Model-Driven Engineering (MDE) is increasingly gaining acceptance, mainly owing to its ability to tackle software complexity and improve the software productivity [5] [6]. MDE promotes the systematic use of models in order to raise the level of abstraction at which software is specified and to increase the level of automation in the development of software. Although the objective of the most common MDE approaches, e.g. MDA [7] and domain-specific development [8], is to build new software systems (forward engineering), MDE has also shown its potential as a reverse engineering technique. Specifically, metamodelling and model transformations have proven to be useful in the automation of many basic activities in software evolution processes, such as representing source code [9] and data [10] at a higher level of abstraction or obtaining information such as metrics [11].

The aim of our work has been applying MDE techniques with three different purposes: (1) tackling the improvement of the logical schema quality in a relational database migration scenario; (2) approaching the integration of a database engineering tool with other software tools; and (3) building a tool able to automate the development of migration processes. By means of the two former, we have investigated the benefits of MDE with regard to traditional solutions, and by the latter objective provided us insights on how MDE may be useful in developing tools supporting software processes. Next, we motivate these objectives.

1.1 Motivation

In recent years, some model-driven reengineering (a.k.a. model-driven modernisation) experiences have been reported [12] [13] [14] and tools with which to support model-driven modernisation are now emerging [15]. Moreover, the OMG launched the initiative Architecture-Driven Modernisation (ADM) initiative with the aim of providing standard metamodels with which to represent information commonly managed in modernisation tasks and thus promoting in this way the model-driven modernisation [14] [16]. However, the focus is on software reengineering (i.e. code) while model-driven *data* reengineering has been overlooked. It is also worth noting that the data engineering community have paid little attention to the application of MDE so far. Therefore, to the best of our knowledge, there are not enough case studies which have been applied to data reengineering approaches and have provided assessment that illustrate the benefits of using MDE techniques in this area.

Data reengineering involves the transformation of legacy artefacts (i.e. schema, data and code) into artefacts of the target system. Transformational approaches have therefore traditionally been used to automate data reengineering tasks such as normalisation, schema conversion, or schema integration as explained in detail in [17][18]. The implementation of such tasks could be facilitated by MDE. The rationales should be sought in MDE, which provides specific technologies (principles, techniques and tools) with which to build *transformational solutions* such as metamodelling and model transformations. Metamodelling foundations are well established and a sound theoretical underpinning is emerging for model transformations [5]. With regard to the techniques, it is well worth mentioning that metamodelling provides a uniform formalism that can be used to represent any kind of information [19] which promotes interoperability and integrability among other quality software factors [20]. And finally, MDE tooling (e.g., Modelling Project in Eclipse) provides (i) metamodelling languages (e.g., Ecore) that can be used to create metamodels and (ii) model transformation languages with which to write model-to-model transformations (e.g., ATL and QVT) and model-to-text transformations (e.g., Acceleo and Xpand).

Referential integrity is an important quality attribute of data stored in relational information systems. It refers to the state where data stored in related tables obeys to the foreign key (FK) constraints defined between those tables. Most modern database management systems purposed for business applications provide features for declaring and automatically enforcing FK constraints. However, many legacy information systems do not use these features - or use them only to a limited degree, for more recently developed functionality. Therefore, it is not unusual that foreign key constraints are left *implicit*, i.e., they are not explicitly declared in the DDL[1] code of the database. Several reasons can be identified: the lack of background in database design in the software development teams [21], the limitations of the target database platform, or the necessity to (temporarily)

[1]DDL stands for Data Description Language

tolerate data inconsistencies [22].

Such applications must be reengineered in order to benefit from automated integrity enforcement. From a high level perspective, this reengineering process consists of two subsequent steps, namely *FK identification* and *FK implementation*. Research activity in this area has primarily focused on the first step (identification), which can be viewed as a form of *design recovery*. A wealth of different methods and tools have been proposed to recover FKs from a variety of data sources, including the database schema, application code, data instances, and documentation. While many FK identification methods have been proposed, empirical evidence about their comparative effectiveness in real-world industrial settings remains rare. Even less research has been devoted to the second step of the reengineering process, as the implementation of FKs (once identified) is commonly considered a simple and straight-forward process.

On the other hand, building tools for supporting MDE software processes is a challenge that must be met to achieve the industrial adoption of MDE [5]. As Leon J. Osterweil stated in his influential paper [23] about the nature of software processes, *"software processes are software too"*, so they can be described by specifications (i.e. models) that can be executable. *Process Engineering* [24] is the Software Engineering area focused on the modelling and enactment of process models. MDE techniques can significantly leverage this area as some works recently presented have illustrated. Most of the activity have been focused on the SPEM metamodel [25] and the approaches proposed to enact SPEM models such as UML4SPM [26], xSPEM [27] and eSPEM [28]. However, the number of proposals that illustrate how real software projects would be supported by using MDE process engineering is very limited up to now [29]. MDE software processes integrate automated tasks (e.g. model-to-text transformations) with tasks to be manually performed by developers (e.g. writing code for the business logic layer. A process engineering tool supporting such processes should provide basic functionality such as: i) the specification of the software process, ii) the execution of the automated tasks, iii) the support and guidance for the software managers and developers involved in the manual tasks, and iv) the integration of manual and automated tasks into a task workflow. To our knowledge, no MDE approaches to the implementation of such tools has been published so far, although the integration of manual and automated tasks have been addressed in building a build server prototype [29].

Finally, data reengineering processes can take advantage of existing tools. For instance, DB-Main [30] includes tools for data reengineering such as: extractors of legacy database schema, transformations between schemas, data and code analysis tools, data viewers, among others. Another example is the Concept Explorer [31] tool which uses FCA (Formal Context Analysis) for identifying functional dependencies in relational databases [32]. The functionality offered by these tools could be integrated into model-driven data engineering solutions by building interoperability bridges. In [33], *interoperability* is defined as *"the ability of two or more systems or components to exchange information and to use the information that has been exchanged"*. System interoperability deals with syntactic aspects (i.e. formats which are used by software systems in order to store data) and semantic aspects (how systems interpret data), as stated in [34].

1.2 Problem statement

This thesis is mainly focused on applying MDE techniques to a data reengineering process which is covered through three dimensions: schema conversion, data conversion and program conversion. This thesis focuses on the first dimension of data reengineering. In particular, we analyse to what extent the use of models facilitates the implementation of the data quality improvement of a legacy system by means of a schema conversion, which is a common data modernisation scenario. The schema conversion implemented in our approach addresses the elicitation of implicit referential integrity constraints (declared in database by foreign keys) along with checking and fixing the appropriate normalisation level in a schema. Several techniques for discovering foreign keys should be combined in order to obtain more reliable results. Furthermore, an automation of migration processes is tackled. We have built a tool that supports the definition and enactment of migration processes, which have been validated for the data migration case study. In addition, MDE solutions normally require the integration with a third-party tool which allows an automatic normalisation step. This requirement leads us to develop an architectural solution to ease tool interoperability and then to integrate other useful tools (from the data engineering and requirement areas) to the migration process here proposed.

From the statement of the problem we can therefore infer the following objectives of this thesis:

(G1) An implementation of a data reengineering process by using MDE techniques. As we stated before, our data reengineering process chases the data quality improvement in a legacy system by means of a schema conversion. As proof of concept, it has dealt with the absence of explicit foreign key constraints in the schema along with the deactivation of constraints. Also, an automatic checking of the database normalisation level in the relational schema is supported. Functional dependencies should be harvested from the database automatically along with a checking of the normalisation level of a database, and a correction if needed. The approach should be validated in a real case study. In addition, we should provide an assessment of the use of MDE techniques, along with a list of benefits and drawbacks found in each stage of the reengineering process.

(G2) The use of different strategies in order to elicit foreign keys for the restructuring stage of the process. A schema conversion analyses a legacy application to extract the source schema and transforms it into a target schema [35]. In our case, the transformation consists of modifying the relations in a legacy database in order to avoid data redundancies and inconsistencies. We should address the discovery of referential integrity constraints which are not explicit in a schema, as well as the checking of the appropriate normalisation level. For the former, three kinds of analysis must be implemented and tested: data analysis, schema analysis and static code analysis, taking into account several kinds of sources and techniques.

(G3) Building a tool able to automate the development of model-driven reengineering processes. We will create a tool specially intended to partially automate model-driven reengineering process. According to the requirements exposed above, this tool should support the definition and enactment of migration processes. A domain-specific language should be therefore created to define migration process models and these process models should be enacted by generating automated and manual tasks. Then, automated tasks could be directly executed and manual tasks could be provided on a task management tool in order to save effort to developers and team leader. With this tool, the approach defined as goal G1 could be automated.

(G4) To tackle the MDE-base tool interoperability through the building of some bidirectional bridge. An infrastructure based on the use of model-driven interoperability techniques must be provided in order to facilitate the integration of useful tools, such as DB-Main and Concept Explorer, into our data reengineering process. A bridge must be defined and implemented in order to allow bidirectional interoperability between two tools or processes. The bridge must be applied during the accomplishment of the main goal in order to facilitate the implementation of the normalisation step (for the identification of functional dependencies).

1.3 Research Methodology

Because the Oracle company discontinued its Oracle Forms technology, many companies have been migrating their applications to modern platforms such as Java EE or .NET. The ModelUM group started a research project to investigate to what extent an MDE-based solution could automate such migrations [36]. For this purpose, it was necessary to develop a tooling for automating the migration of the GUI and data layers. This was the starting point of the research of the PhD candidate.

In order to achieve the objectives of this thesis that were introduced above, we have followed the design science research methodology (DSRM) described in [37] [38]. The design process consists of six activities:

1. Problem identification and motivation.
2. Define the objectives of a solution.
3. Design and development.

4. Demonstration.

5. Evaluation.

6. Conclusions and communication.

This is an iterative process, where the knowledge produced throughout the process by constructing and evaluating new artifacts served as feedback for a better design and implementation of the final solution.

Figure 1.1 shows the DSRM activities applied in this work. Figure 1.1(a) depicts the process followed at the starting point of the thesis. The main problem was identified and motivated in the first stage. The main goal that we established was the definition of a model-driven reengineering process not only for technological modernisation but also for the schema conversion, which includes a data quality improvement. Sections 1.1 and 1.2 stated the problem and motivated the necessity of providing the schema conversion for legacy databases.

Next, we studied the related works, presented in Chapter 3, which referred to the discovery of FK and MDE-based data reengineering processes. We then devised then our solution and the requirements to be satisfied to achieve the objectives of such a solution. In particular, we found that a data reengineering approach should be developed along with an assessment of the use of MDE techniques in this scenario.

In the third activity, the stages comprising the data reengineering process were designed, including the strategies to elicit the foreign key and the techniques and algorithms to perform the normalisation stage (goal G2). First, we experimented with a partial implementation of the process where only one raw strategy to elicit foreign key was provided and code generation was applied. Later, the process was completed and new strategies were integrated into it, as well as the last code generation step. Lastly, we introduced the normalisation step for checking and normalising the schema. In this step we encountered a new problem and we decided to explore the area of model-driven interoperability for data engineering tools. Database normalisation requires the identification of functional dependencies in the relations in order to check the normal form of the schema. To automate the discovery we used a third-party tool. Thus we had to integrate a tool supporting the identification of functional dependencies into our process by using an

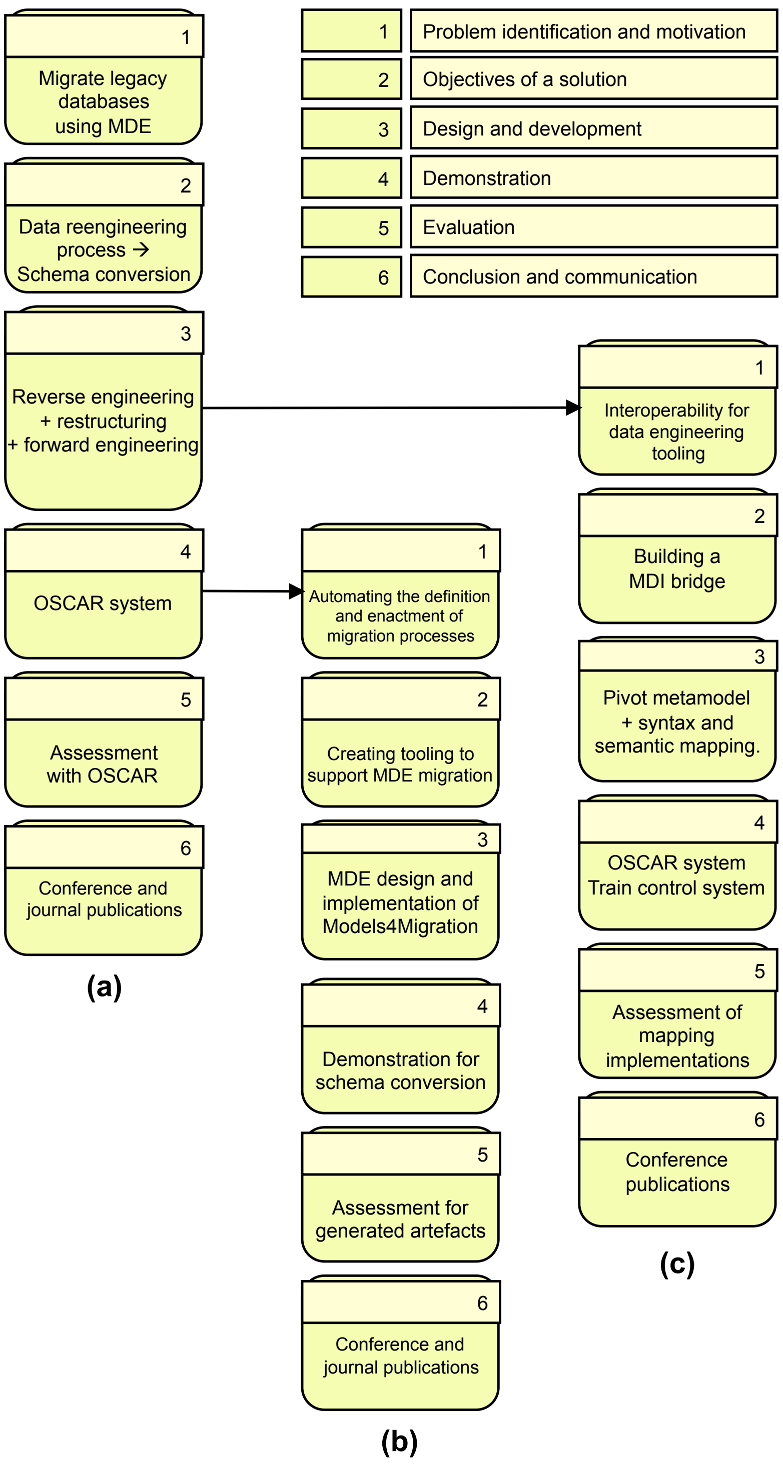

1
Problem identification and motivation
2
Objectives of a solution
3
Design and development
4
Demonstration
5
Evaluation
6
Conclusion and communication
1
Migrate legacy databases using MDE
2
Data reengineering process → Schema conversion
3
Reverse engineering + restructuring + forward engineering
4
OSCAR system
5
Assessment with OSCAR
6
Conference and journal publications
(a)
1
Automating the definition and enactment of migration processes
2
Creating tooling to support MDE migration
3
MDE design and implementation of Models4Migration
4
Demonstration for schema conversion
5
Assessment for generated artefacts
6
Conference and journal publications
(b)
1
Interoperability for data engineering tooling
2
Building a MDI bridge
3
Pivot metamodel + syntax and semantic mapping.
4
OSCAR system Train control system
5
Assessment of mapping implementations
6
Conference publications
(c)

Figure 1.1: Research methodology used in our work.

MDE bridge. Figure 1.1(c) illustrates the six DSRM activities applied in this new research corresponding to the G4 goal.

Once we implemented the proposed reengineering approach, we applied the solution implemented in a real case study. During the 3-month research stay that the PhD candidate did in the Precise group (University of Namur, Belgium), we could access the real legacy database of the OSCAR system, a healthcare application widely used in Canada. For applying our process in the case study we had to investigate a new research line: in particular the building of an MDE-based tool for supporting the automation of model-driven reengineering processes. Figure 1.1(b) shows the DSRM activities designed for the G3 goal.

We conducted an assessment of the approach from the results achieved in the application of our process to the OSCAR system, our case study. In the last activity, we extracted the conclusions and we disseminated the results in conferences and journals. Both conclusions and results are included in Chapter 9 of this thesis.

We summarise in Figures 1.1(b) and (c) the activities followed to fulfill the goals G3 and G4, respectively. Regarding G3, the solution involves the creation of a SPEM-based DSL to define migration processes as well as the design of an interpretation process to enact the migration models defined with this DSL. This solution was demonstrated and evaluated by defining a migration model for the reengineering approach defined to migrate schemas.

With respect to G4, we first built two partial bridges in order to integrate two data engineering tools into our solution: (1) ConExp, a tool for discovering functional dependencies, and (2) the DB-Main data engineering tool. The former enabled the normalisation implementation and the latter allowed us to integrate data engineering functionalities into our data reengineering process. We also built a bidirectional bridge to integrate DB-Main with the Objectiver requirement engineering tool. In the design and implementation, we explored several MDE techniques and tools for injecting models and extracting software artifacts from models. In addition, we addressed the definition of bidirectional model-to-model transformation with QVT relational [39]. We demonstrated and validated the bridges by means of the OSCAR database schema and the traffic control system, a well-known example in the goal (requirement) engineering area.

1.4 Outline

The structure of the rest of this document is as follows:

- **Chapter 2** introduces the background needed for a better understanding of this thesis. It comprises basic concepts of database related to this thesis (such as foreign keys and normalisation), features of a data reengineering process, basis of model-driven engineering and model-driven interoperability, and finally, we characterise the definition and enactment of a migration process (including descriptions about software process languages such as SPEM or BPMN).

- **Chapter 3** analyses the state of the art following the four goals of this thesis: elicitation of foreign keys, MDE data reengineering, migration process tools and MDE tool interoperability.

- **Chapter 4** outlines our proposal for applying a data reengineering process which provides data quality improvements by means of a schema conversion. We also outline the migration tool supporting the definition and enactment of the process, as well as the model-driven approach implemented in order to provide interoperability tool to the process.

- **Chapter 5, 6, 7 and 8** describe in details how each goal of this thesis is achieved and present an assessment and evaluation of the results obtained in the fulfillments of the goals.

- **Chapter 9** concludes this thesis by analysing the level of achievement of the goals we presented in Chapter 1 and the requirements enumerated in Chapter 4. A discussion of each result is presented along with their contributions. We end by showing the future works and the publications, projects and research stays related to this thesis.

"Reek, my lord. Your man. I'm Reek, it rhymes with sneak."
"It does. When my father brings you back, I'm going to take another finger. I'll let you choose which one."

A Dance with Dragons, G.R.R. Martin
(Suggested by Antonio and Paquita)

2 Background

This chapter introduces the background needed for a better understanding of this thesis, which consists of: the definition of basic relational database concepts such as the foreign key constraint and the normalisation process, a characterisation of data reengineering, the basis of Model-Driven Engineering and Model-Driven Interoperability, concerns about the modelling and enactment of migration processes, and finally some considerations for addressing migration processes with MDE.

2.1 Database concepts: Referential Integrity Constraints and Normalisation

In relational databases, referential integrity constraints (RICs) are usually declared by means of *foreign keys*. A foreign key is a simple and intuitive construct through which a row in a table can reference another row in another table. Let us recall the definition of foreign keys in the classical (first Normal Form) relational model.

Considering a table S with primary key KS on the one hand, and a set FR of columns of table R on the other hand, if $R.FR \longrightarrow S.KS$ is declared as a foreign key, then for each row $r \in R$ (that is not null) there should exist a row s in table S such that $r.FR = s.KS$. In other words, the set of values of FR that appears in table R must be a part of the set of values of KS of table S. The foreign key FR acts as a reference to the rows of S.

Database normalisation is a formal analysis of relational schemas for detecting and solving problems in relation schemas. Anomalies in a schema that imply redundant data is one of the most common schema problems. A key concept in relational schema design theory is the functional dependency, which is defined as a constraint between two sets of attributes from the database [40]. A *functional dependency* (denoted by X ->Y) between two sets of attributes X and Y included in a relation (table) R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint restricts that any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$ they must also have $t_1[Y] = t_2[Y]$. This means that the values of the Y attributes of a tuple in r depend on (or are determined by) the values of the X attributes; alternatively, the values of the X attributes of a tuple uniquely determine the values of the Y attributes. The abbreviation for functional dependency is *FD*.

Functional dependencies are used to identify redundant definitions in a relational schema. The *normalisation process* takes a relation schema and applies to it a series of tests to check whether it satisfies a certain normal form. Three normal forms are proposed: first (1NF), second (2NF), and third (3NF) normal form. A stronger definition of 3NF, named the Boyce-Codd normal form (BCNF), was proposed later [40]. All these normal forms are based on a single analytical concept: the existence of functional dependencies among the attributes of a relation. Database normalisation can be considered as a process of analysing a given relation schemas through their FDs and primary keys to achieve the desirable properties of (1) minimising redundancy and (2) minimising the insertion, deletion, and update anomalies. Unsatisfactory relations that do not meet certain conditions (the normal form) are decomposed into smaller relations that possess the desirable conditions. Therefore, the normalisation process provides a formal analysis in order to normalise each individual relation to any desired degree. De-

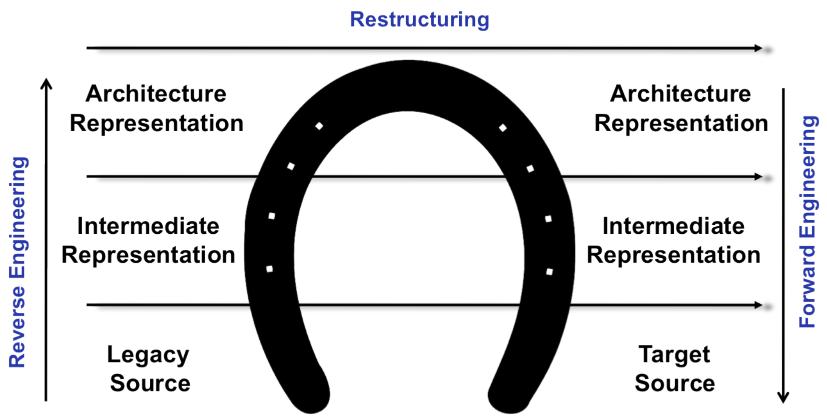

Restructuring
Architecture Representation
Architecture Representation
Intermediate Representation
Intermediate Representation
Legacy Source
Target Source
Reverse Engineering
Forward Engineering

Figure 2.1: Model-driven reengineering stages (extracted from [42])

composition and synthesis algorithms [40] were designed in normalisation theory to be applied alternatively in order to assure that each relation in a schema is, at least, in 3FN. Decomposition applies a top-down approach by decomposing and setting up a relation according to the FDs in a relation as described before. Synthesis applies a bottom-up approach by setting up new relations according to the FDs in a schema.

2.2 Characterising Data Reengineging

Software reengineering is a systematic way of modernising (e.g., a migration) a legacy system [2]. A reengineering process is normally applied in three stages [41], as shown in Figure 2.1. Firstly, a reverse engineering stage analyses the legacy system and extracts knowledge which is represented at different abstraction levels. A second stage restructures these abstract representations in order to establish a mapping between the existing system and the target system. Finally, a forward engineering stage obtains the target artefacts from the output of the restructuring stage. Each of these stages consists of a transformational process.

A data-intensive software system consists of programs that access business data stored into databases. A reengineering approach for the migration of these systems is described in [17], in which the migration process is separated into a synchronised conversion of three different artefacts: data, schemas and programs. Since the changes to the schema affect programs and data, the transformations involved in

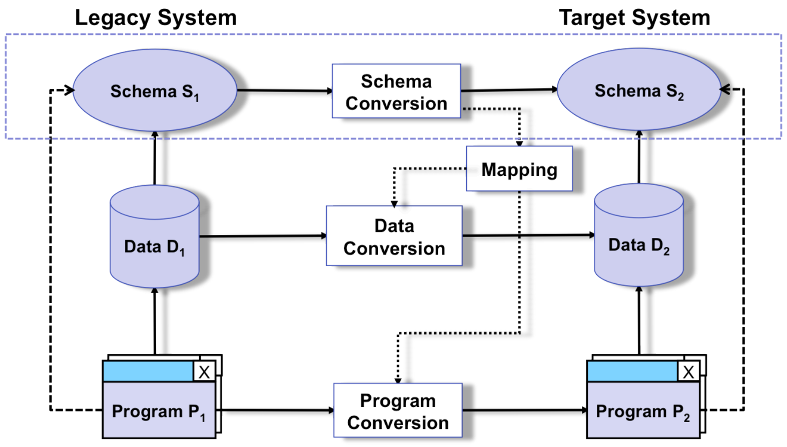


Figure 2.2: Schema conversion in a Data Reengineering (extracted from [17])

this evolution must be coupled as illustrated in Figure 2.2.

Firstly, a semantic-preserving transformation must be applied to the legacy schema in order to obtain a new schema for the target technology. Once this target schema has been generated, data and program transformations are applied next. A new database that is compliant with the new schema is generated from the legacy database. Likewise, legacy programs are modernised for the new database schema. Note that both data/schema and program evolution can be implemented by means of a reengineering strategy.

2.3 Basis of Model-Driven Engineering

Transformational approaches have traditionally been proposed for the automation of data reengineering tasks [18] [17]. Metamodelling and model transformations are the core elements of MDE [34] and they are used to automate tasks by means of transformational solutions (i.e. model transformation chains). MDE may therefore be very useful as regards implementing data transformation strategies. In this section, we shall explain how metamodels and the different kinds of model transformations can be applied in order to implement a reengineering process, such as that shown in Figure 2.1. This explanation facilitates the understanding of the model transformation chains presented throughout the article, which implement the three stages of the proposed approach. We shall, however, first briefly introduce the metamodel and model transformation concepts.

A metamodel is a formalism that is used to describe the structure of models so that *they can be processed by tools.* A model is therefore an instance of a metamodel, and the *conforms-to* term is normally used to express the *instance-of* relationship between a model and its metamodel [34]. Metamodels are defined by means of meta-modelling languages such as Ecore [43] and MOF [44],], which provide the basic constructs of the object-oriented conceptual modelling. In the case of model-driven data reengineering, metamodels can be used to represent information such as data schemas (e.g. a DDL script and a conceptual schema) or knowledge extracted from the data or programs (e.g. Defects model or Functional Dependencies model).

An *MDE solution* is a model transformation chain which generates target software artefacts from input models. Two kinds of model transformations are commonly used in a transformation chain [45]: model-to-model (M2M) transformations which generate a target model from a source model, and model-to-text (M2T) transformations which generate some kind of textual information (e.g. source code or a XML document) from a source model. A model transformation chain normally consists of one or more M2M transformations and one M2T transformation as its last step. The M2M transformations are intermediate stages which reduce the semantic gap, while the final M2T transformation generates the software artefacts desired. In the case of a model-driven reengineering, a third kind of model transformation is needed. In this scenario, the chain should start with a text-to-model (T2M) transformation that obtains the initial model from the legacy textual software artefacts (e.g. source code or DDL scripts). This initial stage is commonly known as *model injection*, which consists of a parsing followed by a model generation.

M2M and M2T transformations are normally expressed using model transformation languages. ATL [1] and QVT [2] are two widely used M2M transformation

[1]https://eclipse.org/atl/

[2]http://www.omg.org/spec/QVT/1.1/

languages, whereas Acceleo [3] and MOFScript [4] are two popular templates languages with which to write M2T transformations. Since M2M transformations can become very complex, the M2M languages should combine declarative and imperative constructs. We have therefore used the RubyTL [5] language, which is a M2T transformation language that is embedded in Ruby whose declarative part is inspired by ATL and whose imperative part is reused from Ruby for free (i.e., any Ruby construct is valid in RubyTL).

With regard to model injection, some tools which have been specially tailored for this task have recently been proposed in order to inject models from source code, in particular the Modisco framework [46] and the Gra2Mol T2M language [47]. Modisco is a Java framework whose aim is to support model-driven modernisations tasks. For example, it facilitates the implementation of model injectors (discoverers), and it currently provides discoverers for Java, JSP and XML. On the other hand, Gra2MoL is the only T2M language available to our knowledge, which has been used in our work in several tasks such as injecting models from DDL scripts and Java source code. This language allows us to write transformations whose execution has as input a textual artefact which conforms to a grammar (e.g. source code) and generates as output a model which conforms to a target metamodel. Gra2Mol provides a language which allows us to express the mappings between a source grammar and a target metamodel. Gra2MoL is a language based on ATL-like rules which allows to express the mappings between a source grammar (ANTLR format) and a target metamodel (Ecore format). In addition, it provides an XPath-like language to declare queries on the syntax tree of the grammar. These queries are used to retrieve the grammar elements to be used in the mappings. Another tool created for injecting models, in this case for data stored into database, is Schemol [48]. This tool provides a language with which to write transformations that express the correspondence between the database schema and the target metamodel. Textual domain-specific languages definition

[3]https://eclipse.org/acceleo/

[4]http://eclipse.org/gmt/mofscript/

[5]http://rubytl.rubyforge.org/

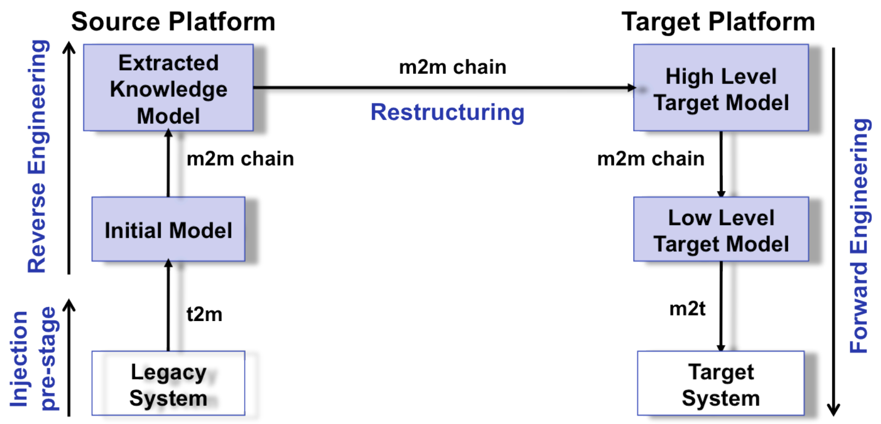


Figure 2.3: Model-driven reengineering stages

tools such as Xtext [6] and EMFText [7] can also be used to inject models, but they are less appropriate in the case of GPL code as discussed in [47].

As depicted in Figure 2.3, a model-driven software reengineering process is composed of three stages. Firstly, a reverse engineering stage analyses the legacy system and extracts knowledge which is represented at different abstraction levels. A second stage restructures these abstract representations in order to establish a mapping between the existing system and the target system. Finally, a forward engineering stage obtains the target artefacts from the output of the restructuring stage. Each of these stages consists of a transformational process which can be implemented by means of a model transformation chain whose initial model is injected with a T2M transformation. This chain is divided into the three parts of a reengineering process: (1) reverse engineering, which is realised by means of an M2M transformation chain; (2) restructuring, which also involves an M2M transformation, and finally (3) forward engineering, which is made up of an M2M chain followed by a final M2T transformation that generates the target artefacts. In the case of data reengineering, three kinds of artefacts (data, schema and program) should be addressed in each stage. We have used RubyTL, Gra2MoL, Schemol and MOFScript to write the model transformations, but the approach is independent

[6]http://eclipse.org/Xtext/

[7]http://www.emftext.org/

of the implementation technologies. It should be noted that data knowledge can be harvested from data stored, schemas and program code.

Finally, models are commonly expressed by means of Domain-Specific Languages (DSLs). Such languages normally consist of three components [49]: a metamodel that describes the abstract syntax of the language, i.e. the set of language concepts and their relationships; the notation or concrete syntax of the language, which can be graphical, textual or hybrid (in the case of graphical notation, the *modelling language* term is often used instead of DSL); and the semantics, defined either by means of a transformation to another language with a well-defined semantics or an interpretation process. Since the abstract syntax of metamodelling languages such as Ecore or MOF is defined by a metamodel, then a metamodel is an instance of the meta-metamodel of a metamodelling language (i.e. a *conforms-to* relationship is also established between a metamodel and a metamodelling language).

2.4 Basis of Model-Driven Interoperability

The development of software systems commonly involves the need to integrate tools. A kind of integration is the exchange of data between tools [50]. Such an integration is called tool interoperability and it is defined as the ability of two tools to exchange information so that the information generated by one tool can be used by the other [33] [50]. Because the exchanged data are represented and interpreted in a different way in each tool, building an interoperability solution (commonly known as a bridge) normally requires the implementation of syntactic and semantic mapping. MDE techniques, especially metamodelling and model transformation, are appropriate to implement such bridges, and Model-Driven Interoperability (MDI) is one of the recognised application scenarios of MDE [34].

Interoperability normally requires addressing both *syntactic* and *semantic mappings* because each tool or component to be integrated can represent the exchanged information in a different format and can also give a different meaning to this information. MDE facilitates the implementation of tool interoperability through the use of metamodelling and model transformation. Models and metamodels provide a high-level representation for the exchanged information, and they act as a lingua franca between the tools [34]. Model transformations ease the implementation of

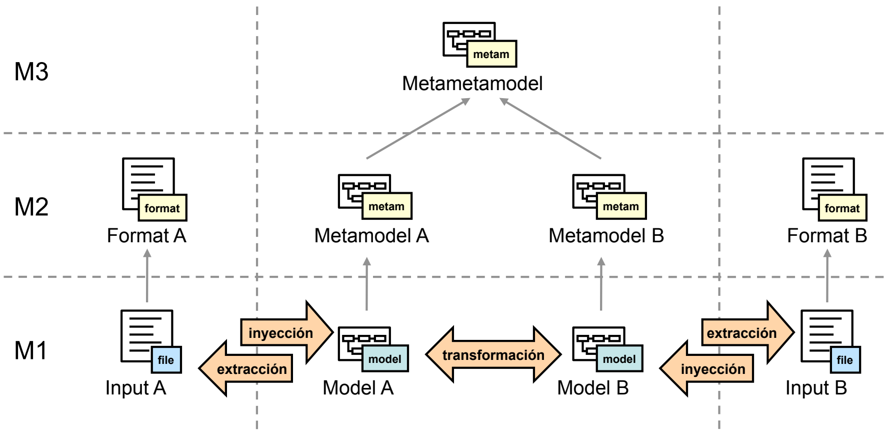


Figure 2.4: Applying MDI techniques between two systems (extracted from [34]).

the necessary mappings. In addition, existing MDE tooling can automate some implementation tasks.

MDE facilitates the implementation of tool interoperability through the use of metamodelling and model transformation. Models and metamodels provide a high-level representation for the exchanged information, and they act as a lingua franca between the tools [34]. Model transformations ease the implementation of the necessary mappings. In addition, existing MDE tooling can automate some implementation tasks. Figure 2.4 shows the elements of a generic MDI bridge for tools A and B, each one using its own data format. Applying MDE in order to build an interoperability bridge involves defining a pivot metamodel for each tool, which represents the concepts underlying to the information managed by the tool. In Figure 2.4, metamodels A and B are the pivot metamodels and both share the same meta-metamodel (e.g. Ecore). Once the pivot metamodels are created, the syntax and semantic mappings can then be implemented. A bidirectional transformation between the pivot metamodels performs the semantic mapping between both tools. If the bridge is bidirectional then two model-to-model (M2M) transformations should be implemented, one for each direction, unless the M2M transformation language supports bidirectionality. Since each tool uses its own format to represent data, the semantic mapping requires a syntactic mapping which consists of two processes: i) the injection process: the exchanged data by the source tool must be converted into the input model to the M2M transformation, and ii) the extraction process: data to be used by the target tool are generated from the model outputted by the M2M transformation. Therefore the building of an MDI bridge implies the following four development tasks:

- *Creating the pivot metamodel for each tool.* These metamodels should not be created if the two tools support exportation of the exchanged information to a metamodel and both metamodels have been defined with the same metamodelling language (e.g. Ecore). If the metamodels have been created with different metamodelling languages then an interoperability at level of meta-metamodel must be defined. An approach to bridge Ecore metamodels and metamodels created with DSL Tools is described in [51]. Actually, most existing tools do not support exportation to models, and those that export some kind of information to models use Ecore metamodels.

- *Creating injectors.* An injector obtains a model, which conforms to a metamodel, from data expressed in another technology (e.g. XML or source code). In MDE interoperability, the injectors are used to represent the source tool data in the form of models that conform to the source pivot metamodel. An injector performs two tasks. Firstly, it parses the source information expressed in the format used by the tool (e.g. a proprietary format or XML). Then, it creates the model that conforms to the source metamodel. How injectors are built depends on the data format. Most tools use a textual format to represent information (e.g. XML or file format). Therefore, the injectors must normally implement text-to-model transformations. Several strategies for building injectors for different formats are analysed in Chapter 8.

- *Creating semantic mappers.* If the bridge is unidirectional any M2M transformation language could be used. However, if the bridge is bidirectional, a language supporting bidirectionality should be used to write only one transformation instead of writing one for each direction. QVT Relational is a good option for this purpose, although the bidirectionality is still an issue open in QVT [52].

- *Creating extractors.* An extractor performs the opposite operation to an injector. In MDE interoperability, the extractors are used to obtain data usable by the target tool from a model that conforms to the target pivot metamodel. As indicated above, most tools represent the information in textual format. Therefore extractors are normally implemented by means of model-to-text transformations. Several strategies for building extractors for different formats are analysed in Chapter 8.

2.5 Modelling and enactment of migration processes

A software process involves the accomplishment of a workflow of activities which create the software artefacts of the target system. Each of these activities can be composed of several tasks which indicate how to fulfill them. For example, the migration of procedures that implement business logic can be done by performing an automatic translation by some means, or by a development team that implement them by hand. Tasks can be classified accordingly to the way they are accomplished in three categories:

- *Automated tasks*: the goal of the task can be achieved without any human intervention, usually by the execution of one or more tools. For instance, the execution of model transformations by means of a model transformation engine.

- *Manual tasks*: the goal of the task must be achieved by a human, either because it is difficult to automate or because it requires supervision. For instance, a code completion task where a developer has to implement some functionality.

- *Semi-automated tasks*: the goal of the task is achieved in a partly automated way as it requires human performance or interaction at some point. For instance, an assistant window which requires some data from an developer to complete some functionality.

It is interesting to differentiate that activities show *what* to do, and the tasks show *how* to do it. Therefore, note that activities are more abstract concepts than tasks. It is also worth noting that activities as well as tasks must be arranged in order for the process to be analysed or executed.

A *process model* specifies the workflow of activities indicating the artefacts that are input and output to each of tasks. Such a model is expressed in terms of types of tasks (e.g. a M2M transformation), tools (e.g. a M2M transformation engine) and artefacts (e.g. files with extension .atl or .ecore). Thus, these models include symbolic names or parameters to refer to tasks and artefacts. They must therefore be *instantiated* prior to be enacted, i.e. the types of tasks, tools and artefacts must be replaced by actual elements (e.g. the path /project/ migration/class2relational.atl and meaningful names). We will use the *Abstract Model* and *Concrete Model* terms to distinguish between a process model and its instantiation for a particular application. Note that both models are at the same level of abstraction, only that a Concrete model assigns values (information on actual elements) to the parameters (i.e. types of elements) included in the Abstract model.

- *Abstract Migration model*: it is designed for a particular pair of source and target technologies. It defines the tools and artefacts involved in the migration and also specifies the order in which the different tasks must be applied.

- *Concrete Migration model*: it is a kind of instance of the *Abstract Migration model* for a concrete migration application.

Enacting a process model involves the execution of the automated tasks and the support for managers and developers in performing the manual and semi-automated tasks. In addition, the enactment of a process model also requires organising its tasks according to the order and dependency relationships among them (i.e. the workflow). In the approach here presented, we have integrated a ticket management tool (Trac) and a task management tool (Mylyn) as target platform of the enactment. Both kinds of tools are commonly used in software companies, and Trac and Mylyn are also plugins of Eclipse, which is a very popular Integrated Development Environment (IDE).

When a development process is aimed to migrate a legacy application, the activities workflow results in a migration process which defines how the existing code and data will be migrated, as well as the deployment approach for the new system.

We will refer as *migration plan* to the specification of a migration process by using a determined formalism. In our work, metamodelling is the used formalism and we will refer to migration plans as *migration models*. The differentiation between abstract and concrete models will be also done for migration plans.

In order to tackle any migration, two higher-level activities must be performed: i) the design of the migration plan itself, and ii) the execution of the tasks specified in the plan. We can suppose that these two activities will be assigned to three different roles: the former for *designer* and the latter for *software manager* and *developer*. Designers specify the workflow of the migration plan, i.e. the flow of tasks, the input and output artefacts for each task and the agents that perform the tasks (tools or developers). Software managers initiate the migration project to enacting it, track the tasks pending to be done and assign them to developers, whereas developers are in charge of fulfilling the manual and semi-automated tasks. Automated tasks are always executed by tools.

Therefore, a tool supporting a migration process must provide functionality for these three roles. Designers require some facilities to define migration plans. Managers require support for enacting migration plans and monitoring task execution. Developers require support and guidance for implementing manual and semi-automated tasks. Moreover, the tool itself must be able to manage the sequencing of tasks following the dependency restrictions established in a migration plan, determining which ones must be performed at a given time.

When addressing migration with MDE, two issues related to the migration and process modelling are introduced: the *implementation* of the different types of tasks in a model-driven migration and the *modelling* of processes. These issues will be further addressed later in this thesis.

We have previously identified three kinds of tasks in a development process: automated tasks, semi-automated tasks and manual tasks. In the case of a model-driven migration, the automated and semi-automated tasks would be implemented using MDE technologies. For instance, an automated task could be implemented as a model transformation chain since it does not require human interaction at all. Some examples of semi-automated tasks are the use of interactive models transformations [53] which prompt the user for input during their execution and the use of wizards to configure the execution of a model transformation through the user input. Finally, the programming of source code either from scratch or from automatically generated code is a manual task. Our re-engineering approach combines these three kinds of tasks.

Actually, these three kinds of tasks can be part of any software development regardless of MDE techniques are or not used. However, in model-driven migration processes, a model transformation chain is normally applied to a large number of existing artefacts. The automation of this repetitive and laborious task means a considerable saving of effort to developers and managers. A tool could locate the files to be migrated and execute the model transformation chains by applying the automated tasks and generating tickets for manual tasks, as indicated above. This particularity motivates the focus of the here presented tool on migration processes instead of software processes in general. Nonetheless, our approach can be easily generalised to any kind of software process.

2.5.1 Software Process Languages

A software process modelling language (i.e. a DSL) provides means for describing the workflow of a software process, that is, the set of activities and tasks that must be executed during the enactment of that process and the order and dependency relationships among them. For instance, SPEM (*Software and System Process Engineering Metamodel*) [25] is a language for process modelling defined by the OMG. The SPEM metamodel has been implemented both with the Ecore and MOF metamodelling languages. Figure 2.5 shows an excerpt of the SPEM 2.0 metamodel, including some of the most significant elements. Below we briefly introduce this metamodel by providing the knowledge needed to understand the extension of SPEM proposed in this work.

SPEM is composed of two main parts: the *Method Content* and the *Process Structure*. In Figure 2.5, the top of the figure shows the `ProcessStructure` package and the bottom shows the `MethodContent` package. There is a clear separation of method content definitions from their application in concrete development processes (e.g. SCRUM), so a method content is a reusable element that can be applied in different processes. The Method Content part provides elements that define elements of a software process, regardless the actual placement

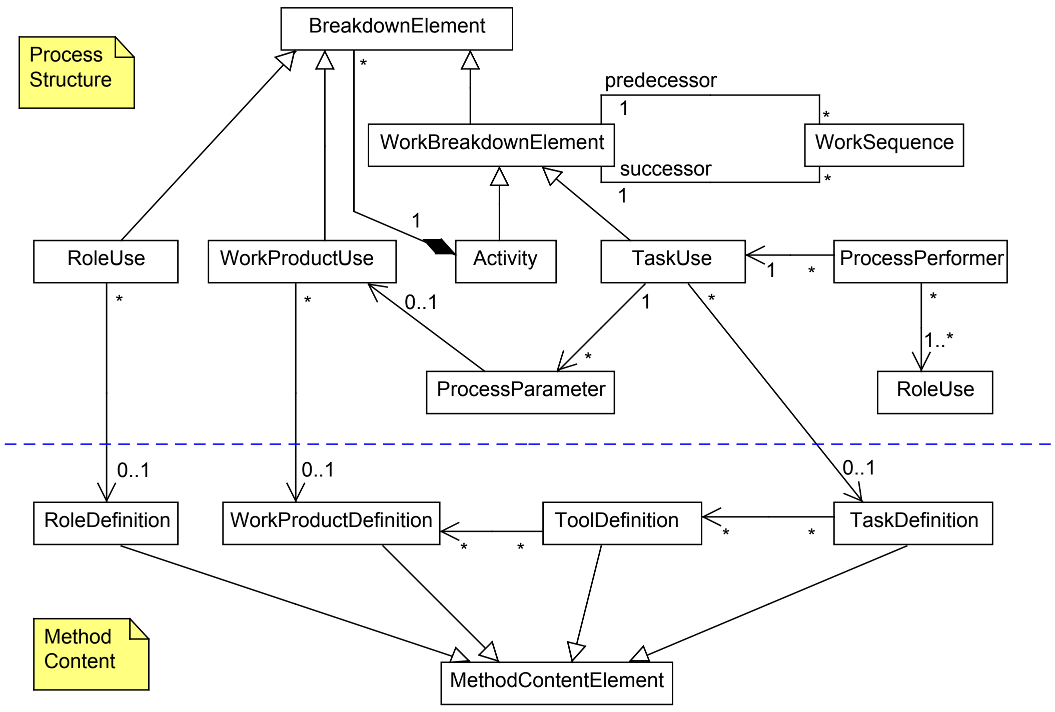


Figure 2.5: Fragment of the SPEM v2 metamodel

of that element in the whole development life-cycle. As observed in Figure 2.5, such elements are, for instance, `TaskDefinitions`, `WorkProductDefinitions` (i.e. tangible or non-tangible artefacts consumed, produced, or modified by tasks), `ToolDefinitions` (i.e. software tools), and `RoleDefinitions` (i.e. roles played by people participating in a software process). `TaskDefinitions` are associated with the `ToolDefinitions` used to accomplish the tasks, and `ToolDefinitions` are associated with the `WorkProductDefinitions` involved.

The Process Structure part defines elements that use the ones defined by the Method Content and relate them into partially ordered sequences that are customised for specific kinds of projects. For instance, for each one of the mentioned `MethodContent` elements, `ProcessStructure` includes the corresponding class that represents an use of these general definitions in a particular process (e.g. `WorkProductUse` and `TaskUse`). In addition, it includes elements to represent `Activities`, and its sequencing `WorkSequence` (i.e. the workflow of activities). An `Activity` also represents a software process itself.

The following relationships are defined in this package:

- `TaskUse` and `WorkProductUse` are related by `ProcessParameter`, meaning that a task has several parameters of the work product type.
- `ProcessPerformer` relates `TaskUse` and `RoleUse` to explain which process roles are involved in a task.
- `Activities` contain `TaskUses`, `WorkProductUses` and `RoleUses`; this is used in our approach in order to define functionality for each activity.
- `WorkBreakdownElements` are sorted by means of `WorkSequences`, so activities and tasks can be sequenced.

With regard to the process enactment, SPEM does not provide its own behaviour modelling concepts. SPEM rather defines the ability for implementers to choose the generic behaviour modelling approach that best fits their needs (e.g. UML activity diagrams). Two strategies are suggested in the specification to enact processes: i) mapping SPEM process models to project plans and then enacting them with project planning and enactment systems such as Microsoft Project, and ii) linking process models to business flow or execution languages and then enact them with a workflow engine (e.g. a WSBPEL engine).

`ControlFlow_ext` and `Activity_ext` are proxy elements in SPEM that can be used to link external process models to SPEM models, in particular, to elements of the Process Structure package. As shown in Figure 7.3, we have used these proxies to connect UML2 Activity Diagrams to our SPEM-based metamodel.

As indicated in Section 2.5, when modelling software processes is needed to distinguish between the definition of a process and its application in a particular case. This is achieved in SPEM by means of classes that represent concrete elements, for instance `TaskUse` for concrete tasks and `WorkProductUse` for concrete artefacts. However, the SPEM metamodel does not provide elements to define the processes at the detail level needed to achieve a high degree of automation (e.g. executing M2M or M2T transformations or creating Trac tickets). For example, an instance of `TaskUse` has links to input and output types that define `WorkProductUses` to be managed in the task.

SPEM provides an extension mechanism based on the `Kind` metaclass. A `Kind` instance is a user-defined type, which can be used to represent a process specific concept (e.g. a model transformation in a model-driven migration process). Therefore, a new type of element can be defined by linking a `Kind` instance to a SPEM element that must be a subclass of the `ExtensibleElement` abstract metaclass. For example, a code file artefact could be represented by linking an instance of `WorkProductDefinition` to a new kind named `CodeFile`. In this sense, the specification defines a `Base Plugin` with kinds of several elements, for instance the `Whitepaper` and `Report` kinds for the `Guidance` extensible element. In order to extend the SPEM metamodel, an option is to use this extension mechanism to define new types for elements of the *MethodContent* package. The problem of this approach is that new attributes or relationships cannot be defined for the new types. For example, if we defined a `TaskDefinition` with a new `Kind` named `M2MTransformation`, but additional attributes such as the `transformationLanguage`, which would register the M2M transformation language used, could not be defined.

"I went to the woods because I wished to live deliberately, to front only the essential facts of life, and see if I could not learn what it had to teach, and not, when I came to die, discover that I had not lived."

from Walden, Henry David Thoreau
(Suggested by Maribel)

"Success flourishes only in perseverance ceaseless, restless perseverance"

Baron Manfred von Richthofen (Red Baron),
Imperial German Army Air Service
(Suggested by Jose Antonio)

3 State of the art

This chapter presents the review of the state of the art that we have carried out to analyse the existing proposals related to our work. This review is organised in four sections according to the main goals of this thesis: foreign key discovering, model-based data reengineering, migration process tool and tool interoperability.

3.1 Foreign Key Discovering Techniques

The problem of undeclared foreign keys elicitation has already been extensively studied and a large variety of techniques have been proposed in the last two decades. [54–64]. Discovering *implicit* schema constructs, especially undeclared foreign keys, is usually based on ad hoc techniques depending on the nature and reliability of the information sources. Several techniques have been proposed, each considering a particular information system artifact as the main source of information:

- *Schema analysis* [65, 66]. Spotting similarities in names, value domains and representative patterns may help identify hidden constructs such as foreign keys.

- *Data analysis* [67–70]. Mining the database contents can be used in two ways. Firstly, to discover implicit properties, such as functional and referential dependencies. Secondly, to check hypothetic constructs that have been suggested by the other techniques. Considering the combinatorial explosion that threatens the first approach, data analysis is most often applied to check the existence of formerly identified patterns.

- *Screen/report layout analysis* [71–73]. Forms, reports and dialog boxes are user-oriented views on the database. They exhibit spatial structures (e.g., data aggregates), meaningful names, explicit usage guidelines and, at runtime, data population and error messages that, combined with dataflow analysis, provide information on data structures and schema properties.

- *Static program analysis* [61, 74–76]. Even a simple analysis, such as dataflow graph exploration, can give valuable information on field structure and meaningful names. More sophisticated techniques such as dependency analysis and program slicing can be used to identify complex constraint checking or foreign keys. SQL statements examination is one of the most powerful variants of source code analysis.

- *Dynamic program analysis* [63, 64]. In the case of highly dynamic program-database interactions, the database queries may only exist at runtime. Hence recent techniques which allow the capture and analysis of SQL execution traces in order to retrieve, among others, implicit referential links between columns of distinct tables.

The next Table 3.1 categorises the indicated approaches according to four criteria: the kind of *analysis*; the validation by means of a *case study*; the existence of tools for *automating* the approach; and the *combination* of results obtained from different kinds of analysis. The last row categorises our proposal.

It is essential to note that none of the above techniques is generally sufficient to recover all implicit referential constraints. In other words, there is no formal way

Approach	Analysis	Case study	Automated	Combinated
[65] Markowitz90	schema	no	manual	no
[66] Premerlani94	schema	yes	manual	no
[67] Chiang94	data	yes (small)	manual	no
[68] Lopes02	data	yes (medium)	manual	no
[69] Yao08	data	yes (medium)	automated	no
[70] Pannurat10	data	yes (medium)	semi-automated	no
[71] Choobineh92	screen	yes (not explained)	automated	no
[72] Terwilliger06	screen	no	automated	no
[73] Ramdoyal10	screen	no	automated	no
[74] Petit94	program (static)	no	manual	no
[75] DiLucca00	program (static)	yes	manual	no
[76] Henrard03	program (static)	no	manual	no
[61] Cleve06	program (static)	yes	automated	no
[63] Cleve11	program (dynamic)	yes	automated	no
[64] Cleve12	program (dynamic)	yes	automated	no
our proposal	schema, data and static program	yes	automated	**yes**

Table 3.1: Related work on FK discovery.

to prove that all the undeclared foreign keys, and only these have been discovered through a particular technique. As often in the field of reverse engineering, automated analysis techniques may only *suggest* possible foreign key candidates with a certain level of confidence. In addition, the availability of a ground truth, allowing the evaluation of a particular detection technique, is not a realistic working assumption, especially in the context of large legacy systems.

3.2 Model-driven Data Reengineering

Model-driven reengineering (a.k.a. Model-driven modernisation) has recently emerged as a use case of MDE, signifying that the research and development effort in this area has, to date, been very limited. Moreover, most publications are focused on the program dimension rather than the data dimension of the software applications. In fact, model-based transformational approaches have traditionally been applied to database development since the 1970s. However, MDE provides a more systematic means to define and manage database schemas, mainly through the use of metamodelling techniques and model transformation languages.

KDM [77] is a common intermediate representation proposed by the OMG for existing software systems and their operating environments, which defines common metadata required for deep semantic integration of Application Lifecycle Manage-

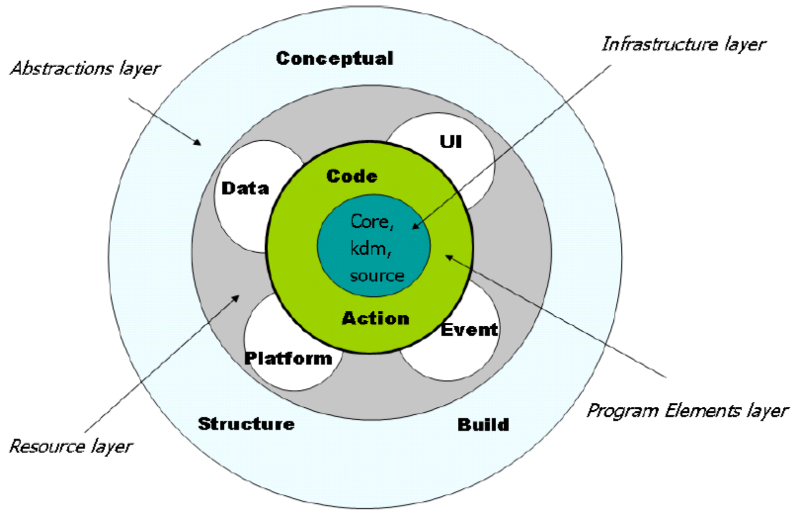


Figure 3.1: KDM layers and packages (extracted from [77]).

ment tools and which was defined for software modernisation in the context of the *Architecture-Driven Modernisation (ADM)* proposal. KDM is a metamodel aimed at representing software systems at different levels of abstraction which range from program elements to business rules. KDM is intended to facilitate the interoperability between software modernisation tools, as a common representation for software artefacts. It is a very large metamodel that is composed of twelve packages organised in four layers: *Infrastructure*, *Program elements*, *Runtime resources* and *Abstractions* (see Figure 3.1). Each package defines a set of metamodel elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems. The packages defined in the specification are:

- **Core** and **Kdm**: define common elements that constitute the infrastructure for other packages.
- **Source**: enumerates the artefacts of the existing software system and defines the mechanism of traceability links between the KDM elements and their original representation.
- **Code**. focused on representing common program elements supported by various programming languages, such as data types, data items, classes, procedures, macros, prototypes, and templates, and several basic structural

relationships between them.

- **Action**: along with the *Code* package, this represents the implementation level assets of the existing software system. This package is focused on behaviour descriptions and control and data-flow relationships determined by them.
- **Platform**: defines a set of elements whose purpose is to represent the run-time operating environments of existing software systems.
- **UI**: represents facets of information related to user interfaces, including their composition, their sequence of operations, and their relationships to the existing software systems.
- **Event**: specifies the high-level behaviour of applications, in particular event-driven state transitions.
- **Data**: is used to describe the organisation of data in the existing software system.
- **Structure**: is aimed at representing architectural components of existing software systems, such as subsystems, layers, packages, etc. and define traceability of these elements to other KDM facts for the same system.
- **Conceptual**: provides constructs for creating a conceptual model during the analysis phase of knowledge discovery from existing code.
- **Build**: represents the facts involved in the build process of the given software system (including but not limited to the engineering transformations of the "source code" to "executables").

The Data package defines a set of meta-model elements whose purpose is to represent organisation of data in the existing software system. This fact of knowledge corresponds to the logical view. It is determined by a data description language.The Data model uses the foundation provided by the Code package related to the representations of simple datatypes. The Data model represents complex data repositories, such as record files, relational databases, structured data stream, XML schemas and documents.

The Data package is a very large metamodel which is extremely complicated to manage as explained in [11]. In our experience, creating one's own metamodels is more convenient than extending KDM with new elements. The metamodel is neither appropriate for modelling sentences extracted from a DML script nor establishing references between DML and DDL elements. In addition, the concept of functional dependency is out of the scope of the metamodel. Summarising, the Data package in the KDM metamodel defines the general elements in order to model a legacy information system (including code and data) but, when more details are needed (as our solution requires) then KDM is hard to manipulate.

We next discuss some of the most relevant works concerning the application of MDE in Data Reengineering and data reverse engineering.

- A conceptual framework for the evaluation and improvement of database schemas is presented in [78], and is based on semantics-preserving transformations and the identification of specific patterns in schemas. This framework deals with database schemas at any level of abstraction and relies on three main ideas: i) the definition of equivalence classes that includes a set of constructs representing the same intention as that of a modeller, ii) the formal definition of the quality requirements, iii) the existence of evaluation and improvement methods. A defect taxonomy has additionally been proposed for this framework in [79], which classifies the design flaws into several categories and establishes semantic-preserving transformations in order to replace a defect with a construct that belongs to the same equivalence class. Our approach could be a good starting point at which to develop the framework and the taxonomy proposed.

- An MDE reverse engineering approach with which to elicit relational database schemas from embedded SQL code is described in [80]. The elicitation process removes the dead parts of the schema which are not used by database programs. This problem is similar to that of removing defects from schemas and we have therefore carefully contrasted this approach with our proposal. A two-stage process is applied to generate the new schema. Firstly, an SQL sentence model is obtained by means of a static analysis of the source code, and a model-to-model transformation is then applied to a reverse engineering process in order to transform this model into a database schema model which represents the schema that is really managed by the SQL code. The reengineering process is confusedly and poorly explained. For instance, the authors use KDM but do not clarify how the KDM models are obtained or how they are related to the SQL sentence model and the database schema model. Nor do they explain how restructuring and forward engineering are performed. Unlike with our approach, the authors have manually developed a parser (i.e. an injector) to inject models (KDM and SQL sentences models) from the SQL source code. We have defined our own metamodels rather than using KDM. Moreover, the description of the approach lacks the level of detail needed to understand how the MDE techniques have been applied, and an assessment on the application of MDE is not provided.

- An MDE approach for data conversion has recently been presented in [81]. Logical and conceptual schemas are inferred, as are the mappings between them. The inferred schemas and mappings are refined by means of consultation with domain experts. These mappings are represented as models that represent Datalog-like queries which are converted in a data flow graph. Moreover, a query translator converts the SQL queries in the source schema into the equivalent query in the target schema. The tool was built in a 1 person year and it has been applied in a real migration scenario that had previously been manually migrated with an effort of more than 5 person years. The estimated gain in productivity is therefore about 80%. The authors do not provide an analysis of the advantages or drawbacks of using MDE techniques in the implementation of the tool presented.

- Some strategies with which to reengineer a legacy information system are described in [18], which addresses both program and data migration. Although a transformation strategy is applied in the schema conversion, MDE techniques are not used, but instead a DDL parser library is provided to extract an ad-hoc representation of the physical schema of the database from certain data-managing languages (such as SQL, Cobol, or XML, among others). Data and programs are analysed in order to obtain the information needed to refine the physical schema generated in a logical schema which is finally transformed into a conceptual schema and stored in a proprietary format named GER model. These transformations are implemented with the transformation toolkit provided by DB-MAIN [82]. The most important downsides of implementing these strategies are the automation and interoperability levels.

- A data reengineering approach is defined in [83] in order to generate a new application in a multilayer system using different programming languages and platforms (JSP, Java Swing, EJB, Windows Forms) from a physical schema of the database. Firstly, a platform independent model is injected from the physical schema. This model is then transformed into a UML-like class model which represents a possible conceptual schema of a database, and this class model is eventually used to generate the new application. The injection step is implemented by reading directly into the data dictionary of a Database Management System (DMS), like our DAS-D strategy. This approach is therefore an ad-hoc solution for each DMS. Implementations for the most common DMS (e.g. Oracle and SQLServer) are included in this work. Metamodels are represented as a Java class model and transformations are implemented by means of Java code. This proposal is more limited than our approach since the defect correction and possible schema normalisation are not considered. Moreover, the implementation does not take advantage of the existing MDE techniques/technologies.

- The automation of the schema normalisation task has been tackled in some works, but a model-driven solution is only considered in [84]. The authors define an approach with which to achieve a conversion between normal forms which is based on (i) the definition of a UML-like class metamodel for expressing database schemas, (ii) OCL statements with which to specify the normal form levels, (iii) and graph-based transformations that can be used to write transformation rules that define the mapping between normal forms. However, they do not address the extraction of functional dependencies from data schemas, which our approach automates by applying the FCA theory and the integration of the ConceptExplorer tool [31].

- DB-MAIN [82] is a toolbox that offers a complete functionality with which to apply data engineering (i.e. from data requirement definition to data evolution and maintenance), which includes tools for data reengineering such as: extractors of legacy database schema, transformations between schemas, data and code analysis tools, data viewers, among others. The development of this tool is the result of a great number of research contributions to the data reengineering area over the last few years by the LIBD laboratory at Namur University. With regard to our work, in the schema transformation they apply a transformation strategy but they do not use MDE techniques and they implement a DDL parser which extracts the database physical schema and then analyses the applications and the data to extract the information needed to refine the schema obtained in a logical schema which is eventually transformed into a conceptual schema. They use a schema transformation toolkit provided by DB-MAIN. The degree of automation and interoperability achieved by means of MDE techniques in our proposal is better than that achieved in DB-MAIN.

Table 3.2 summarises the previous works according to the next features (last row points the values of our proposal out):

- (1) *MDE* - Is this an MDE solution?
- (2) *Stages* - What Data Reengineering stages are implemented?
- (3) *Data* - Are data considered in the Data Reengineering process?
- (4) *Extensible* - Could the approach implementation be extended with new defects/fixings?
- (5) *Metamodel* - Indicates the technology used to define the metamodel.
- (6) *Assessment* - Is some kind of assessment provided in the approach?
- (7) *Automated* - Is the solution automated through a tool or similar?

The previous table denotes that all analysed works present some kind of automation except [78] and [79] which present a conceptual framework aimed at improving schemas quality. All the works (except our proposal) lack some assessment that results from its application to a case study or running example. Only [80] uses the KDM metamodel. However how KDM is applied is not described or some lesson

Approach	MDE	Stages	Data	Ext.	Metamodel	Assessment	Automated
[78][79] Lemaitre10 Lemaitre11	✘	all ?	✘	✔	-	✘	✘
[80] Perez12	✔	all	✘	✘	KDM	✘	✔
[81] Yeddula15	✔	all	✔	✘	UML	✘	✔
[18][82] Hainaut05 Hainaut94	✘	all	✔	✔	GER	✘	✔
[83] Polo07	✘	all	✔	✘	UML	✘	✔
[84] Akehurst02	✔	reest.	✔	✘	UML	✘	✔ (no FD)
our proposal	✔	all	✔	✔	ECORE	✔	✔

Table 3.2: Related work on model-driven data reengineering.

learned on the use of this metamodel.

A survey of software reengineering tools is presented in [85]. The authors review eleven tools which are categorised according to several criteria. Four of them are based on MDE: Blue Age, Modisco, Obeo Agility and Moose. Application migration is supported by Blue Age and Obeo. Modisco is an MDE-based framework that offers KDM model injectors for several languages such as Java and JSP. It is worth noting that only one of the reviewed tools (DB-Main) is intended for data reengineering and most of the reviewed tools are commercial.

3.3 Migration Process Tool

Our work is related to the *Process Engineering* area, which is concerned with the definition and enactment of software process models. In this setting, the works related to our proposal belong to two main categories: SPEM-based approaches to enact software process models and the use of MDE techniques to create process engineering tools. In addition, we also consider Business Process Model and Notation (BPMN) [86] as an choice to define a new DSL for modelling and enacting processes.

3.3.1 SPEM extensions

As indicated in Section 2.5.1, SPEM does not address the enactment of process models, but SPEM models must be linked to behaviour models expressed with other formalisms. This design choice aimed at providing flexibility for SPEM implementors, so that they can select any behaviour modelling formalism. However, a significant research effort has been devoted to extending SPEM to make it a

Process Modelling Language (PML) with capabilities of executing process. The proposed approaches differ in the way of extending the SPEM metamodel, the notation provided to express behaviour models, the formalism used to define the semantics, and the goals of the enactment of software processes. Next we will discuss three of the most relevant proposals: UML4SPM [26], xSPEM [27] and eSPEM [28].

UML4SPM is a metamodel based on UML 2.0 activity diagrams intended to overcome some limitations of SPEM 1.1, in particular the lack of behaviour models and some issues related to the expressiveness (e.g. constructs to model the human interaction during the development of a process and the existence of different work products). An operational semantics (named Execution Model) has been defined for UML4SPM as the basis of the execution support [26]. In [87], a UML4SPM-based framework, which combines the aspect-oriented and MDE approaches, is proposed for modelling and executing process models. In this framework, the process models are made executable by means of execution models expressed in the Kermeta language [88]. Rather than integrating with management tools, the framework supports executing automated and manual tasks as Java programs from Kermeta code. UML4SPM extends the UML activity diagram metamodel with concepts needed to enact software process, such as *Tool*, but we have encountered some limitations in using UML4SPM in our approach. For instance it does not allow the definition of how tasks are executed on tools (e.g. its input parameters, dependencies, etc.). In addition, concepts and notation are less suitable than using a SPEM-based metamodel.

The main goals of xSPEM are the execution and validation of software processes. Process validation is achieved by using translational semantics, i.e. transforming SPEM models into Petri nets and then checking them using a dedicated tool. Process execution is supported by WSBPEL through the definition of a mapping between xSPEM and a WSBPEL extension called BPEL4People [89]. We have not used xSPEM due to the difficult of adapting the enactment provided in our approach. Moreover, the metamodel is complex and it would be necessary to complete and modify the generated code for supporting the process execution.

The extension eSPEM has been mainly designed to solve three shortcomings of SPEM, namely: (i) its lack of behaviour modelling; (ii) it does not support the definition of new tasks when the process is already being executed; and (iii) it does not facilitate the configuration of a process for several methods. eSPEM is based on UML activities and state machines for the execution of processes. The eSPEM operational semantic has been formalised with fUML [90] in order that the tools based on eSPEM can support process validation. Several editors are available for creating eSPEM models and an integrated chain of tools allows the modelling and the enactment of software process models [91]. Once the process models are created by using the editors of the Process Modelling Environment (PME), they can be deployed into a Process Execution Machine (PEX) where a management team can access it to create development projects which are performed by the developers. The PEX offers support and guidance (a sensitive context help for tasks) to the developers and allows the control of the process to the managers by reporting on the tasks performed, deviations and traceability. A significant difference between eSPEM and xSPEM is that the former provides fine-grained constructs of behaviour models as decisions or tasks scheduling. Despite some advantages over UML4SPM and xSPEM, we have found that the eSPEM metamodel does not include the elements needed to achieve the execution desired in our approach, i.e. executing automated tasks and generating tickets which are integrated into a task management tool.

Whereas the existing SPEM extensions aim to offer a general-purpose process modelling language, our approach is specific to a software development domain, in particular software model-driven migration processes. In this setting, our goal was to express a migration process at a granularity level that is appropriate for executing automated tasks (e.g. model transformations) and controlling the execution of manual tasks by integrating tools like Mylyn and Trac. The greater specificity of our approach favours a model enactment that conveys the execution of tasks, since our process models contain all the needed information. The enactment based on the integration of Trac and Mylyn for the manual tasks and the execution of Ant files for the automated tasks is the main characteristic that differentiates our approach. To the best of our knowledge, our approach is the first approach to define migration process models that are enacted for concrete migration scenarios. However, the current version of Models4Migration does not address some aspects that are commonly supported in process-centered environments, such as validation and monitoring.

3.3.2 BPMN LANGUAGE

BPMN [86] is an OMG specification aimed at providing a graphical language for business processes. The language has been specifically designed to coordinate the sequence of tasks in a set of activities, and the messages that flow between the different participants that perform those tasks. The BPMN specification defines a mapping for WSBPEL (Web Services Business Process Execution Language) [92], which is an XML-based language for describing business processes in which most of the tasks represent interactions between the process and external web services. The WSBPEL process itself is represented as a web service, and is interpreted by a WSBPEL engine which executes the process description. Therefore, the enactment of BPMN diagrams can be performed with WSBPEL.

Although BPMN supports process execution in workflow engines by using execution languages such as WSBPEL, we consider that this is not suitable for our approach because BPMN lacks expressiveness in the area of software processes, in particular migration processes, so it would have to be extended with new elements, which will not be available in BPMN editors. Moreover, the mapping from BPMN to WSPBEL would have to be extended to support these new elements. In addition, workflow engines such as the WSBPEL engine cannot be easily integrated with development environments (IDEs). For example, there is no support to communicate the workflow with an IDE in such a way that the IDE shows a list of methods to implement, and the workflow engine updates this when the methods are completed.

In short, we conclude that neither BPMN nor SPEM can be directly used for our purposes. SPEM has been conceived for software processes but it does not provide enactment and can not be easily extended to allow software processes to be expressed at the level of detail required for automating tasks, such as executing a model transformation or generating tickets. BPMN supports the process enactment but we cannot take advantage of this feature since the language is not targeted at software processes. With regard to SPEM extensions, we have found that the defined metamodels do not include the elements needed for our approach and the enactment provided cannot be easily adapted in order to automate and control programming tasks. We therefore decided to create a new DSL whose metamodel is based on SPEM. In particular we have adopted the SPEM core concepts (tasks, roles, work products, tools, etc.) and we have extended them to fit the software migration domain. For enacting the models we have implemented used activity diagrams.

3.3.3 Using MDE in Software Process Engineering

A pioneer work on model-driven process engineering was presented in [93]. A software environment is viewed as a bus where tools are plugged in and models provide a standard interface, i.e. the data managed by a tool is transformed into a model that conforms to the metamodel of that tool. This idea is widely used in our framework and we have gone a step further by defining the concept of cartridge, which holds all the artefacts needed to generate a specific migration environment. Moreover, we have tackled the execution of automated tasks and the control of the manual programming tasks. The emergence of the SPEM language has caused an increased interest in applying MDE in process engineering, and most of the approaches to model software processes are based on this OMG standard. However, these works have not had an impact on the industry yet.

To our knowledge, the automated support of MDE development has been only addressed in [29]. In particular, the authors present a prototype of a MDE-based build server which manages the integration of automated and manual tasks. Automated tasks are automatically executed and users are notified that a manual task should be done. Every time a task is completed, other automated tasks can be triggered and/or manual tasks be notified to users. The scripts allow specify verifications and validations operations on the artefacts created and consumed in tasks. The tool is based on a metamodel that represents artefacts, operations and several kinds of verifications. The *Operation* metaclass includes a boolean attribute to indicate if the task is manual or automated and references to the source and target artefacts. Each operation has two UUID's to indicate the implementation and technology. The *Artefact* metaclass includes attributes whose values are assigned in build time, which provide information on the execution status of the artefact.

Artefacts also include an UUID to indicate its location. This metamodel is used to create build script models which are interpreted by the tool implemented.

Next, we comment on the most significant differences between this build server and Models4Migrations. Our approach is focused on a migration process scenario that needs to apply the same transformation process to a large number of legacy resources many times. Therefore, it would be required to generate many tasks (even more than legacy resources to be migrated), and they should be automatically generated in order to alleviate effort for designers and managers of the migration process. Moreover, in [29] the authors propose their own task and dependency management system instead of using a well-known system, such as Trac, which provides blocking/unblocking of tasks along with a dependency control among them. The Trac server enables the migration managers to define (i) the dependencies among tasks, (ii) the task assignments to developers and, (iii) in general, a complete control flow of the enactment of process tasks. We have integrated Trac and Mylyn which allows the use of the generated tasks in widely used environments such as Eclipse. The approach proposed in [29] supports process partial execution, which allows the execution of tasks even when all their inputs are not provided.

Method Engineering is a software engineering area related to *Process Engineering*, which is focused on the creation and customisation of software methods [94]. Several works have illustrated how MDE techniques, such as metamodelling and model transformations are very useful in this area. For instance, MetaEdit illustrated the concept of a metamodelling-based tool that generates editors supporting method notations sixteen years ago [95]. Although our work does not fit in Method Engineering, it can be related to the proposals for creating method definition tools. For instance, Cervera et al. [96] have recently presented an MDE approach to extend the modelling tool Moskitt [97] with SPEM support in order to provide capabilities for the definition and configuration of software methods and the automatic generation of the tool to support them. The methods are customised to a specific context by linking them to concrete technologies (i.e. editors and model transformations). This approach, unlike ours, provides neither enactment nor guidance to developers.

Regarding the migration tools, traditional tools (e.g., JANUS from Software

Revolution [1]), do not offer enough flexibility to define custom migration processes and enact them. Some model-based migration tools (e.g., Agility from OBEO [2] or the migration tool from BluAge [3]) are emerging, but they use models to improve some aspects of the software quality such as interoperability and productivity, and they do not provide a process modelling language to define custom migration processes.

Task-based development processes such as the one proposed in our approach are commonly used in software construction (e.g., agile methodologies or the development of free and open source software) to identify and perform the required development tasks. Bug-tracking (e.g., BugZilla [4] and Trac [5]) and issue-tracking systems (e.g., Mantis [6]) are normally used with Mylyn (and extensions such as TaskTop [7]) to drive development processes. However, both tool integration and task creation have to be manually performed by developers. We propose a seamless integration with the migration process, as well as the generation of the required migration tasks. A related approach in the context of IDE plugins is proposed in [98]. The authors present a framework that extends IDEs with task-based plug-ins and workflows that automatically execute multiple tools and integrate their results. Compared to this approach, our tool is integrated into an IDE (Eclipse) and it not only provides executing automation applications, but also programming support facilities.

We next summarise in Table 3.3 the tools to support software process that we have presented in this section (last row presents the values of our proposal), according to the following features:

- (1) *Migration* - Is this a migration-specific solution?.

[1]http://www.tsri.com/about-us/tsri-services.html (accessed on 11-20-2014)

[2]http://www.obeo.fr

[3]http://www.bluage.com/en

[4]http://www.bugzilla.org

[5]http://trac.edgewall.org/

[6]http://www.mantisbt.org/

[7]http://tasktop.com/mylyn

- (2) *MDE* - Has MDE been used in implementing the tool?
- (3) *Tasks* - Does it generates manual or automatic tasks to be integrated into an IDE?
- (4) *Cartridge* - Is it an extensible tool?
- (5) *Definition* - Does it support process definition?
- (6) *Instantiation* - Is instantiation or similar supported?
- (7) *Enactment* - Does it provide enactment support?

Approach	Migration	MDE	Tasks	Extensible	Process Definition	Instantiation	Enactment
[93] Bezivin01	✘	✘	✘	✘	✔ (no SPEM)	✘	✔
[29] Steudel12	✘	✘	✔	✘	✔	✘	✔
[96] Cervera10	✘	✔	✔ (tool)	✘	✔ (SPEM)	✔	✘
[98] Mariani13	✘	✘	✔ (semi-auto)	✘	✔ (workflow)	✘	✔
JANUS	✔	✘	✔ (partial)	✘	✘	?	✘
OBEO	✔	✔	✔	✔	✘	✔	✔
BluAge	✔	✔ ?	✘ ?	✔	✘	✔	✔
our proposal	✔	✔	✔	✔	✔ (SPEM-like)	✔	✔

Table 3.3: Related work on migration process tools.

The first four related works compared in the table are not aimed at applying migration process, thus they lack of process instantiation which is a relevant feature in the migration scenario (except [96], that provides a sort of process instantiation but not process enactment). On the other hand, commercial solutions such as JANUS, OBEO and BluAge do not include the ability to define a migration process by means of a process language. Finally, only OBEO and BlueAge (as well as our proposal) were built on an extensible architecture which provides the mechanism for adding new kind of tasks, artefacts and technologies during process definition.

3.4 Tool Interoperability

Tool integration has been a topic of great interest from the early years of software engineering. In [50], tool integration is analysed from several dimensions and more recently a research agenda has been proposed in [99]. According to the definitions proposed in [50], our work is focused on *tool interoperability* In the review of the literature presented in [99], Model-driven interoperability approaches are not considered since MDE was then just emerging and XMI is shown only as a novel format which exchanges data.

The injection and extraction processes needed in a model-driven interoperability are an example of bridging Modelware technology (i.e. MDE) and other technologies (e.g. XML, Grammarware and APIs). The concept of *technical space* was introduced in [100] to define technologies at a high-level of abstraction. This notion was used in [93] to establish bridges between different technologies (e.g. Grammarware, Modelware and Ontologies). Each technical space is characterised by the pair of concepts data/format and the formalism used to define the data formats, e.g. code/grammar and EBNF for Grammarware, and model/metamodel and metamodelling language for Modelware. In this way, mapping between technologies can be established at three different levels: data, format and formalism. Gra2MoL [47] and Api2MoL [101] are examples of tools created in order to build bridges between Modelware and other technical spaces, in particular Grammarware and APIs, respectively. Textual DSL workbenches also bridge Modelware and Grammarware. Support in order to bridge Modelware and XML technology is provided by EMF [43]. In the case of the Objectiver/DB-Main bridge, Modelware can be mapped to three different technical spaces: Grammarware (LUN format), XML and APIs (JIDBM). Therefore, we have considered the above mentioned tools.

We next enumerate the MDI approaches that will be compared to our solution:

- How MDE could be used in the tool interoperability scenario was shown in [102] for the AMMA MDE framework. This work proposed implementing semantic mapping by means of a weaving model created with the AWM (Atlas Weaver Model) tool. The approach was illustrated by means of a bidirectional bridge between two bug tracking tools. A weaving model allows only express links between model elements, and this can only be applied if the mapping is very simple. Therefore, M2M transformations are normally used to implement MDI bridges.

- In [103], another MDI approach is presented for the AMMA framework, which uses an M2M transformation to implement a unidirectional semantic mapping for converting textual reports generated by a clone detection tool into SVG files. AMMA provides a DSL definition tool named TCS (Textual Concrete Syntax), which is used to automatically generate the injector for the clone detection reports.

- A similar approach to the integrations of tools is described in [104] for the MOFLON framework, which uses MOF as metamodelling language. In contrast to these two approaches our work has tackled the building of a bidirectional bridge for EMF that is the most widely used MDE platform. Moreover, we have considered different alternatives for implementing the injector and extractor.

- A bridge for integrating models created with different metamodelling languages, in particular Ecore and metamodelling language used in some Microsoft modelling tools (e.g. DSL Tools) is proposed in [51]. This approach could be used in our work if a platform different to EMF is used to integrate Objectiver with other tools.

Table 3.4 summarises how each MDI stage is implemented in the commented approaches. Again, last row describes our proposal.

Approach	Pivot Metamodel	Syntactic Mapping	Semantic Mapping
[102] Fabro06	Ecore (AMW)	no?	✔ bidirectional ✘ simple mapping
[103] Sun08	MOF	✔ injector autogenerated ✘ unidirectional (no extrac.)	✘ unidirectional mapping
[104] Amelunxen08	MOF (UML)	✘ manual?	✔ TGG (Triple Graph Grammars) ✔ bidirectional mapping
[51] Bruneliere10	several metamodels	✔ XML injector ✘ not extractor yet	✘ unidirectional mapping (ATL)
our proposal	EMF	✔ several alternatives (inj+ext)	✔ bidirectional (QVT)

Table 3.4: Related work on tool interoperability.

The works commented in the previous table are applying MDI to specific problems. Therfore, they are not intended to explore and analyse different strategies with which to implement the mappings of a model-driven bridge. In [103] and [51], the semantic and syntactic mappings are unidirectional and they only provide injectors but not extractors. In [102] and [104], syntactic mapping (injectors and extractors) is not provided and these solutions are focused on providing a transformational bridge between the tool data once they are stored in models. It is worth remarking that [51] is the only MDI approach able to deal with different metamodels.

Applying MDE techniques to automate tasks in the area of requirement engineering has been widely addressed in the literature. Great research effort has been devoted to defining MDE solutions that generate software artefacts from requirement models. For instance, conceptual multidimensional models for data warehouses are generated from i* goal models in [10], security software artifacts (e.g. rules implementing security policies or security code for a database from security models in [105], or KAOS goal models from mind maps in [106]. Note that the work presented here is mainly focused on building MDI bridges, rather than addressing other topics related to the model-driven requirement engineering. However, it is worth remarking that our approach illustrates an MDE application scenario to be explored in this area. That is, how requirement tools may be integrated with other tools in building software systems, and how software artefacts can be generated from requirement models.

"Do not tell me how I can reengineer my mind or life, tell me how I can undo the past"

anonymous
(Suggested by Jesús García)

4 Overview

In previous chapters we have stated four high-level objectives to be achieved in this thesis work through an analysis of the state of the art in model-driven software reengineering. This chapter is devoted to outline the approaches proposed to fulfil the goals. Previously, a set of requirements is elicited for each goal.

4.1 Requirements

The main goal of this thesis was defined in Section 1.2. It consists on *applying MDE techniques to a data reengineering process and analysing to what extent the use of models facilitates the implementation of the data quality improvement in a schema conversion.* This goal is separated into four high level goals: (G1) an implementation of a data reengineering process by using MDE techniques, (G2) the use of different strategies in order to elicit foreign keys for the restructuring stage of the process, (G3) an implementation of a process tool with which to provide the support for the definition and enactment of MDE-based data migration processes, and (G4) building a model-driven bidirectional bridge in order to investigate the integration of external tools into a data reengineering process.

We have elicited a set of requirements for our solution, which can be organised in five groups: the first one defines the general requirements for the work to be developed in this thesis and the other groups are related to the four high-level goals. Next, we will present all these requirements.

Our work of applying MDE to improve tasks of data reengineering has been driven by the following general requirements:

(R1) **Productivity.** Productivity is one of the main factors to be taken into account in the implementation of any software system. Developers of a data reengineering solution have to generate new artefacts on schedule and by using the least resources as possible.

(R2) **Automation.** We are interested in automating our data reengineering process as much as possible so that it can be easily applied our data reengineering process to a large number of legacy systems with minimum effort.

(R3) **Modularity, Extensibility and Reusability.** It would be desirable to split the data reengineering process into simpler stages to make it maintainable. In addition, a solution split in decoupled stages would facilitate extension (for instance, to add new processing stages) and reusability in different projects. The reuse of solutions or partial implementations is mandatory to attain high levels of productivity.

(R4) **Evolvability.** As stated in [4], the integration of software and database evolution processes is one of the current challenges to be addressed by the reengineering community. They usually evolve independently thus leading to inconsistencies resulting in high costs of software and data maintenance.

(R5) **Consistency.** Another challenge identified in [4] is the lack of synchronisation among the database schema changes and code that accesses data by means of an Object-Relational Mapper (ORM). In software evolution, development teams sometimes work in an undisciplined manner (i.e., database evolves without considering the ORM definitions and vice versa)

(R6) Technology Independence. The data reengineering process should be easy to reuse with different technologies (source independence). Furthermore, it must be extensible, so that new target platforms can be added without changing the reverse engineering and restructuring stages (target independence).

With regard to the implementation of an MDE data reengineering process (G1) we have identified the next requirements:

(R7) Representing a database at high abstraction level. Database information related to a data reengineering process must be provided in terms of high-level concepts, such as possible defects or functional dependencies.

(R8) Providing manual support to the migration actors in order to manage the schema conversion. The database administrator's knowledge should be considered in order to decide which of defects detected must be removed from the schema. This kind of solution would be according to human aware [107].

(R9) Variability in the generated artefacts. Multiple kinds of artefacts should be generated in accordance with the migration requirements: scripts for re-generating the database, new middleware code for accessing data or both.

Regarding the use of different strategies in order to elicit foreign keys (G2) we have defined the next requirements:

(R10) Use of multiple sources for the analysis. Database information to be analysed could be harvested from schemas, database records or programs.

(R11) Use of different techniques for the analysis. The same analysis could be implemented using different MDE techniques.

(R12) Combining results from the different analysis. Instead of consider a unique set of results obtained from one strategy, it would be desirable to consider a combination of the results of each strategy.

With respect to the implementation of a tool supporting a model-driven reengineering process (G3) we have established the following specific requirements:

(R13) Supporting for the process definition. The tool should provide a language tailored to the definition of migration processes. Process definition scripts should be injected as models in order to apply model-driven techniques in the implementation.

(R14) Ability to define migration processes in abstract and concrete form. A process definition (i.e. an abstract process model) is independent from concrete artefacts. They use symbolic names instead of concrete references to tools and workproducts. However, the enactment of a process model requires references to the concrete artefacts involved into the execution. Therefore, process models must be instantiated to replace symbolic names by the legacy resources of the system to be reengineered (i.e. concrete process models). This is a specific requirement of a migration scenario which differs from another one.

(R15) Supporting for the process enactment. The tool should enact the data migration process by automatically performing the automated tasks and generating manual tasks to be manually completed (guided if possible).

(R16) Integration with existing development environments. Productivity could be facilitated if the completion of manual tasks could be performed by using complete and mature developing environments (e.g. Eclipse).

(R17) Tasks should define dependencies among them. Manual or automated tasks should represent dependencies among them in order to enact a data migration process. Tasks will be in a blocked or unblocked status. A task could only be executed if all its precedent tasks (in terms of dependency) are unblocked.

(R18) Supporting for task assignment. Managers of a migration process should deliver tasks among the migration developers. The tool must provide support to ease how tasks are assigned and who are the developers in charge of its completion.

Last set of requirements are referred to a model-based architecture in order to allow the tool interoperability (G4):

(R19) **Bidirectionality.** It would be desirable that given two tools A and B to be integrated, data in A format could be used by the B tool and vice versa.

(R20) **Dealing with different scenarios.** In particular the MDE interoperability should be investigated in two scenarios of integration: a third-party tool is integrated into a model-based solution and two tools are integrated being one of them a data engineering tool.

In order to attain the goals stated in accordance with the requirements defined, we proposed (1) the implementation of a model-based data reengineering approach for the schema conversion in a migration along with (2) a tool with which to define and enact the process. The process proposed requires the use of an external tool in order to search for the functional dependencies as part of the normalisation step applied during the restructuring stage. The integration of external tools has been achieved by developing (3) a tool interoperability solution by means of MDE techniques. Finally, we have defined (4) several strategies for the foreign key discovering in the reverse engineering stage of our process, by creating algorithms with which to implement schema, data and static code analysis. We will next outline each of the achievements identified above. In the following chapters, we will describe in detail all of them.

4.2 A Model-based Data Reengineering Process

In Section 1.2, we stated a problem regarding the recovery of implicit constructs (i.e. potential defects in the database schema), in particular, foreign keys and check constraints. Our approach is a model-driven software reengineering process as depicted in Figure 2.3. The outline of the reengineering approach proposed to solve the problem is shown in Figure 4.1. Each reengineering stage corresponds to one of the three tasks into which the problem is decomposed (see Figure 2.3). The defects in the original schema are detected in the reverse engineering stage; the changes made to the schema in order to correct defects and apply a normalisation

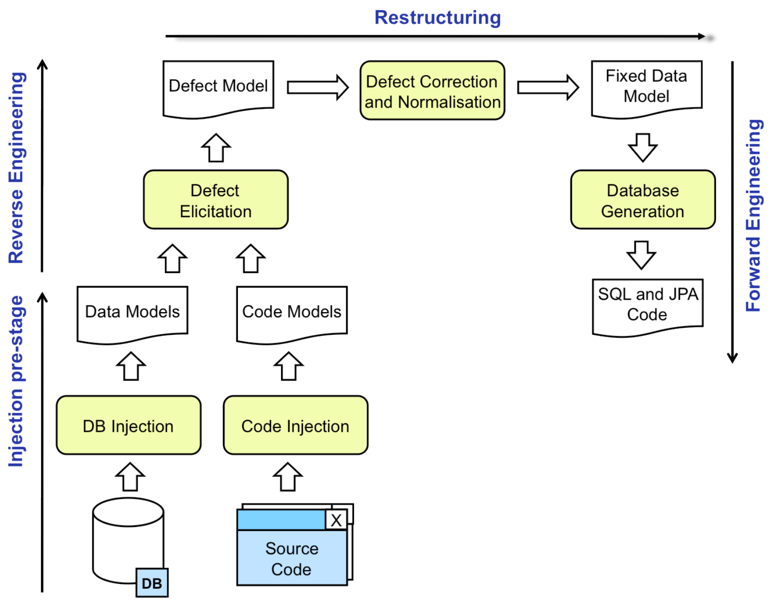


Figure 4.1: Model-based data reengineering process.

process, if needed, correspond to the restructuring stage; and forward engineering techniques are applied to generate the new artefacts of the migrated system. The steps to be performed in each stage are briefly introduced below. We will provide a detailed description of how each of the stages has been implemented by means of model transformations in Chapter 6.

4.2.1 Reverse engineering

Program and data analysis are two complementary strategies that are commonly used to elicit implicit constructs in database schemas [108] [109]. In our case, the data analysis is performed on either data stored in the database or on data models injected from DML (Data Manipulation Language) scripts. Whatever the strategy applied, the database schema must also be analysed; a schema model has therefore been injected from the DDL script. Furthermore, the code analysis is performed on models injected from the source code of database programs. Both strategies are implemented by means of M2M transformations and generate a Defect model which includes the defects to be removed from the legacy schema. The strategies we have implemented in our approach were combined and presented in [108] in collaboration with the Precise research group. Section 6.3 will show how they have been implemented as part of a model transformation chain following an MDE approach.

4.2.2 Restructuring

The legacy schema is firstly modified to remove the defects. This schema conversion is implemented as an M2M transformation. Once the schema has been fixed, an analysis is performed to check whether the normalisation level should be changed or not. Functional dependencies are first identified by using a specific tool and then, a normalisation is applied if necessary. This process is implemented as an M2M transformation chain that integrates the tool used to detect functional dependencies. The result of this stage is therefore a fixed and normalised schema (Fixed Data Model in Figure 4.1). This stage will be explained in Section 6.4.

4.2.3 Forward engineering

Finally, the source code of target artefacts is generated from the model obtained in the previous stage. Two kinds of artefacts are generated: SQL scripts with which to regenerate the database and the code of a middleware that is used to access the new schema. JPA is the standard technology to implement the persistence layer of a Java application. Our approach generates the Java code and configuration files according to the new schema resulting from the restructuring stage. This process is also implemented as a model transformation chain, which will be described in Section 6.5.

4.3 Strategies for the Foreign Key Discovering

The data reengineering process, which is defined in this thesis, deals with the discovering of referential integrity constraints (RIC) not declared in relational schemas, along with the detection of constraints disabled (foreign keys and check constraints). We have implemented two strategies that extract the knowledge from different sources: database and program source code. The first strategy, which is based on database information, is composed of two analyses that have to be applied orderly: firstly a database schema analysis and then a data analysis. The second strategy, which is based on program code, uses a static code analysis of the SQL sentences embedded in the source code of the database applications. The implementation of the two strategies are included as part of the reverse engineering stage of our data reengineering approach and they will be described in Chapter 5.

We have also defined a manual process to tackle the RIC detection in a legacy system through the joint analysis of multiple sources of information: the database schema, the database contents and the program source code. The results obtained by each analysis technique are then combined in order to find a certain number of *likely* foreign key candidates. This work was carried out in collaboration with the Precise group at the University of Namur during the predoctoral stay of the PhD candidate.

4.4 A Tool to Define and Enact Model-based Migration Processes

In this section we will outline the design choices we have taken to satisfy the requirements of the tool built to support MDE-based migration processes. The tool has been implemented by applying MDE techniques.

4.4.1 Process definition

A modelling language (i.e. a DSL) is a desirable feature of the tool in order to make labour of specifying software processes easier to designers. We have found that SPEM is not suitable for modelling software processes when enacting and interacting with external tools is required (i.e. model transformation engines), due to the following reasons:

- *Lack of detail*: the concepts provided by SPEM work at a high abstraction level, while our approach needs to define processes at a low-level of detail for the execution of tasks, what implies including information such as resource paths or configuration options.
- *Lack of enactment support*: SPEM does not support the enactment of process models since it does not provide execution semantics. SPEM strategies

to provide enactment support are not suitable since no mapping for workflow engines is provided and project management tools do not really execute process models.

Therefore, SPEM as-is cannot be used for our purposes since we want it to include MDE domain concepts and additional information to allow enactment. It is worth noting that the extension mechanism described in 2.5.1 is not appropriate as new attributes and relationships cannot be defined for the new types. Since that several approaches have been presented to extend SPEM in order to enact software processes, we have analysed some significant proposals to assess if they are able of supporting the enactment of our approach. This analysis is summarised in Chapter 3 where we indicate the limitations of these extensions to support the kind of enactment proposed in our approach. We have therefore chosen to create a new DSL to define migration process. This language is based on SPEM and lets defining the elements (e.g. tasks and workproducts) and workflow related to migration plans, as well the information required for the enactment. The metamodel used to define the abstract syntax of the DSL will be referred to as *Migration metamodel* and will be described in Section 7.3.1.

4.4.2 Process instantiation

As explained in Chapter 2, a Concrete Migration plan must be instantiated from an Abstract Migration plan in order to perform the enactment for a particular application. Both Abstract and Concrete Migration plans are represented in our approach as models, namely the *Abstract Migration model* and the *Concrete Migration model*, respectively; both models conform to the Migration metamodel. An Abstract Migration model is created by the designer for a pair of source and target technologies and it is specified in terms of variables that denote some kind of elements or artefacts involved in the migration (e.g. a JPA Java class or a DDL script). A Concrete Migration model is the result of instantiating an Abstract Migration model for a particular application to be migrated. This instantiation consists of replacing each variable of the Abstract Migration model by its corresponding concrete artefacts (e.g. actual resources with real names and paths), which are part of the source application.

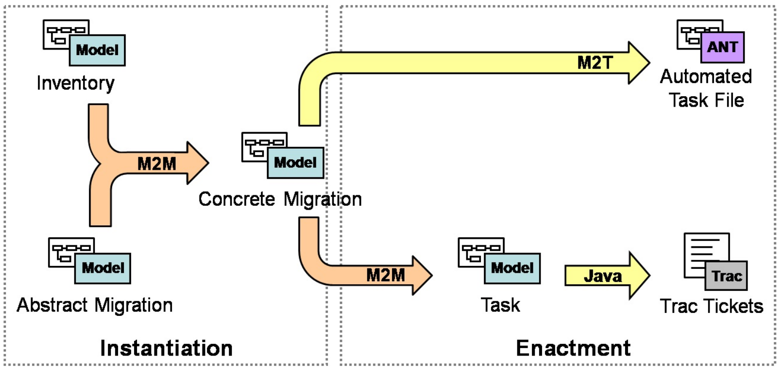


Figure 4.2: Instantiation and enactment of migration processes.

The instantiation of an Abstract Migration model into a Concrete Migration model is implemented as an M2M transformation which has the Migration metamodel as its source and target metamodel. In addition, this transformation also uses an *Inventory* model that provides the input artefacts that are needed to instantiate a given Abstract Migration model. Therefore, the transformation has an Abstract Migration model and an Inventory model as its inputs, and generates a Concrete Migration model, as shown in Figure 4.2.

4.4.3 Process enactment

The instantiation creates an enactable process from a process definition. Each task of an enactable process is interpreted and carried out by an agent (person or machine), depending on if the task is manual or automated. A tool supporting process models must provide a high-level of automation in the execution of enactable tasks. In our approach, two model transformation chains automate the enactment of a concrete migration plan, as illustrated in Figure 4.2. This enactment is achieved by the execution of the automated tasks and the support of the manual tasks. The execution of automated tasks is done by using the Ant build tool. The Ant file to be executed, namely *Automated Tasks* file, is generated by an M2T transformation which input is the concrete migration plan. With regard to the manual tasks, the tool generates information that can be integrated into management tools. The support is currently provided by the integration of the Trac bug server with the Mylyn tool, which is a contribution of our work.

We use Trac tickets to represent pending tasks which are integrated into Mylyn, an Eclipse plugin for the management of tasks, which configures the user interface of the Eclipse IDE showing only the resources that the developer needs to fulfil a certain task. The Trac-Mylyn integration is based on an M2M transformation which generates a *Task* model from the *Concrete Migration* model, and a Java process which creates the tickets into the Trac server by means of querying the Task model by EMF API and creating the tickets by the Java Trac API.

4.4.4 Technology-Independence

A tool supporting software processes should be independent of the implementation technologies used. In our case, an independence of source and target technologies is achieved by using models to define the migration plans and the way they are interpreted. Models4Migration is a cartridge-based tool implemented as an Eclipse plugin. A `Migration Cartridge` is a pluggable module that encapsulates an *Abstract Migration model* (i.e. the migration process), a *Process Interpretation model* (i.e. the definition of the behaviour to be performed when enacting the migration plan), and all the resources required to enact the Abstract Migration model. Both models conform to the Migration Plan metamodel and they are expressed in terms of the source and target technologies.

While an Abstract Migration model specifies the workflow for a migration (e.g. Oracle Forms to Java), a Process Interpretation model specifies how to interpret this model in order to extract manual and automated tasks to be integrated into the process management tools. Therefore, the Process Interpretation model describes the chain of model transformation shown in Figure 4.2, although extensions to this chain may be needed depending on the pair of specific technologies, as we will explain in Section 7.5.

The basic process executed by the Models4Migration tool is described in Figure 4.3. A *Process Interpreter* has the Inventory model and the models encapsulated into the Migration Cartridge as its inputs and generates the *Task model*, which defines contexts (explicit information about the tasks and resources) for the tasks defined in the Concrete Migration model, and the *Automated Task file*, which is executed by the Ant tool creating a new Eclipse project with the resources

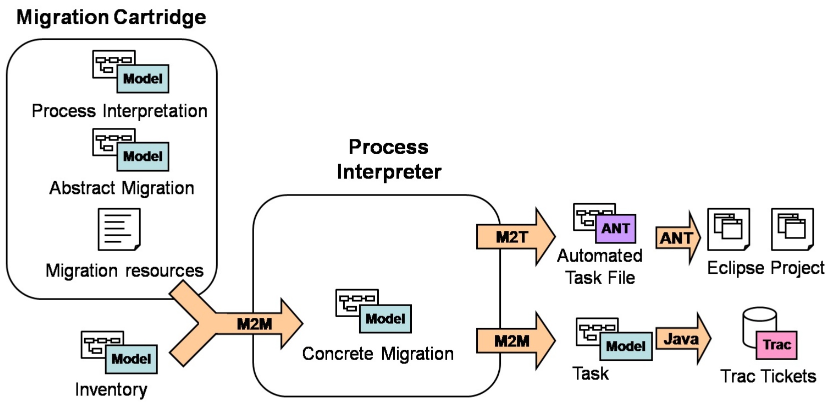


Figure 4.3: Process Interpreter in the migration process tool proposed.

generated by those tasks. The Task model reduces the semantic gap between the Concrete Migration model and the Trac API. This model is interpreted by a Java program to create tickets on a Trac server.

Transformations shown in Figure 4.3 were implemented by means of the facilities included in the AGE environment [110], using the RubyTL transformation language for the M2M transformations and the MOFScript language for the T2M transformation.

The sections in Chapter 7 describe in more detail the elements introduced above and their application to the running example. We will start with a section that analyses the suitability of existing process model languages to describe software processes in a way appropriate to the kind of enactment applied in our approach.

4.5 A Tool Interoperability Architecture

As indicated in Section 2.4, tool interoperability normally requires addressing both syntactic and semantic mappings because each tool or component to be integrated can represent the exchanged information in a different format and can also give a different meaning to this information.

In this thesis, we firstly tackled the tool interoperability issue through the implementation of a bridge between the Concept Explorer tool and our data reengineering process. Afterwards, we implemented another bridge between the DB-Main tool and our process. In this second case, we experimented with several strategies for creating injectors and extractors, in order to compare them. Both bridges only implement a syntax mapping to integrate such tools in the MDE technical space. Hence, we implemented a third bridge aimed to integrate the DB-Main and Objectiver tools, which is shown in Figure 4.4. In this bridge, database schemas and KAOS object models are the information to be exchanged. Since Objectiver imports/exports KAOS models from/to an Ecore metamodel, we have just defined a pivot metamodel which represents the database schemas in DB-Main. Therefore, the creation of an injector and an extractor is not needed for Objectiver. Instead, three kinds of injectors have been created for DB-Main, taking into account the three options provided by this tool which can be used to access its information: a Java API named JIDBM, a XML file generated by an exportation plugin and the DB-Main project file (*.lun* files). As shown in Figure 4.4, for each option two implementation strategies have been considered. For the JIDBM API we have used EMF API and the Api2Mol tool [101]. For the XML file we have used EMF API along with JAXP [1] and JAXB [2], and on the other hand, the XML serialiser of EMF. Finally, for the project files we have explored the use of the Gra2Mol tool and DSL workbenches, such as Xtext [3] and EMFText[4]. QVT Relational has been used to implement the semantic mapping.

4.6 A Case Study: the OSCAR system

This section will introduce the main case study used in our schema conversion process and the strategies for FK discovering. It is worth remarking that the case study used in the migration process tool is our own migration process.

Our case study is an information system that is widely used in the health-care industry in Canada, called OSCAR (Open Source Clinical Application Resource) [111]. OSCAR is a so-called Electronic Medical Record (EMR) system.

[1]http://jcp.org/en/jsr/detail?id=63

[2]http://jcp.org/en/jsr/detail?id=222

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

[4]http://www.emftext.org/index.php/EMFText

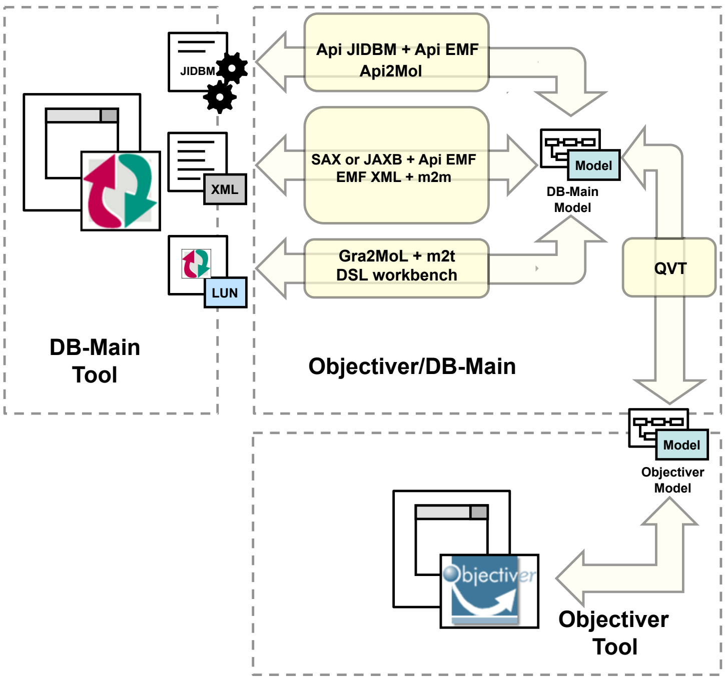


Figure 4.4: Objectiver/DB-Main bridge.

Its primary purpose is to maintain electronic patient records and interface with a variety of other information systems used in the healthcare industry, such as laboratory systems, pharmacy systems, specialist information systems, etc. It also provides advanced functionality related to typical clinical workflows, e.g., electronic prescribing functions. OSCAR has been developed since 2001, originally by the Department of Family Practice at McMaster University. Today it is one of the most popular EMR systems in Canada and in use by thousands of GPs across the country. As the acronym suggests, OSCAR is open source software. Current development activities on OSCAR are coordinated by a not-for-profit company ("OSCAR EMR"), under an ISO 13485 certified development process.

OSCAR has been using MySQL as its DBMS platform. MySQL supports the choice of different alternative storage engines. During the first five years of OSCAR development, MySQL did not support a storage engine capable of enforcing referential integrity. Consequently, OSCAR's database implementation does not make significant use of FK constraints but rather consists of seemingly unrelated tables. Over the last several years, OSCAR has been migrating to MySQL's newer InnoDB storage engine, which provides full support for referential integrity enforcements. Since then, more recently developed parts of the system have made use of FK constraints. Still, the vast majority of the database tables remain without any explicit relationships in the schema. This situation has been a frequent source of frustration in the OSCAR developer community as it impedes program understanding and maintenance. It has also raised concerns with respect the integrity of patient health information and, ultimately, patient safety. Therefore, it has been a goal to reengineer OSCAR with respect to establishing more referential integrity constraints.

The database was originally supported in a MySQL system but it was also recreated in an Oracle database, in order to apply some of the analysis implemented in this thesis. As it is said below, the database used is encrypted for the sake of privacy.

We encountered a number of *challenges* in our case study which affect directly to the FK discovering proposed in our solution. The most important ones are pointed out here.

- *Size.* One obstacle in this process is the sheer size of the database schema. With close to five hundred tables and some of the larger tables comprising over thousands of columns, identifying FKs cannot be a manual process but requires automated tool support.

- *Multi-paradigm architecture.* Another challenge is the unevenly evolved nature of the OSCAR architecture, which uses a multitude of different paradigms to access the database. Some older application modules still use embedded (dynamic) SQL queries, while newer modules use object-relational middleware descriptors (Hibernate mapping files), and yet newer application code uses code annotation tags based on the Java Persistence Architecture (JPA) standard. Therefore, no single method for detecting FKs in application code is likely to recall all relevant relationships.

- *Confidential data.* Knowledge about the actual database instances is an important prerequisite for the process of identifying RICs. It is not uncommon that the data in legacy information systems is considered business confidential. However, patient records are among the most sensitive and highly regulated information items in any industry and they cannot commonly be made available for the purpose of software engineering, even under non-disclosure agreements. We had to create software and a process to securely encrypt the data prior to FK analysis and attain approval from the University ethics board prior to our reengineering study.

"L'amitié est une religion sans dieu ni jugement dernier. Sans diable non plus. Une religion qui n'est pas étrangère à l'amour. Mais un amour où la guerre est la haine sont proscrite, où le silence est possible"

Tahar Ben Jelloun
(Suggested by Nesrine)

"These are the two pillars of a Trappist life. There has to be a balance between work and monastic life. We earn our living. There's no reason to change that, or make more money"

Brother Joris, Westvleteren brewery
(Suggested by Loup)

5 Strategies of Defect Discovering

This chapter will be devoted to presenting the strategies used to elicit the foreign key constraints applied in our approach. We shall describe these strategies from an algorithmic point of view, i.e. without dealing with implementation details. The following sections will not therefore include explanations about the MDE techniques used to implement the algorithms.

As noted in Chapter 4, in this thesis we have implemented three analyses in order to discover the FK. The first two analyses used the legacy database as a source and had to be applied in a orderly manner: firstly a database schema analysis and then a data analysis. The third analysis was based on taking the program code as source and used a static code analysis of the SQL sentences embedded in the legacy applications. Our three analysis techniques will be described in the following sections, after which two more brief sections will be devoted to outlining the two analysis developed by the Precise group. We shall conclude the chapter by showing the triangulation of all the results and discussing it.

It should be noted that the triangulation of the results was realised manually and it is no implemented inside our data reengineering process. Moreover, we will just outline the last two analysis techniques (Hibernate and JPA analysis) because they were not implemented for this thesis but they were developed by members of the Precise group and more details can be found in [108]. We have just included the results obtained by these two analysis in order to show how they were combined along with the results obtained from our analysis (schema, data and static SQL analysis).

5.1 Schema Analysis

The *Schema Analysis* process is guided by the primary key constraints found in the tables of the schema. Each column $columnPK$ contained in a primary key of a table is used to search for other columns in the database schema that could reference $columnPK$. Algorithm 1 specifies this process. We use *tablePK* and *columnPK* variables to refer to the table and the column, respectively, of the primary key side.

Algorithm 1 *Schema Analysis* algorithm

```
$result \leftarrow \emptyset$
for $tablePK \in schema$ do
    for $columnPK \in tablePK.constraintPK$ do
        $result \leftarrow result + SearchForFK(tablePK,$
                                    $columnPK)$
    end for
end for
return $result$
```

In the *SearchForFK* function, columns are searched based on their names and data types (SQL type, length and precision) as we show in Algorithm 2. Variables *table* and *column* are used to refer to the table and the column, respectively, analysed as a candidate foreign key.

The *EqualsNames* function returns *true* if the names of the columns and tables analysed are compatible, and returns *false* otherwise. Algorithm 3 explains in a precise way what *'compatibility'* means in this context, and shows how the function works.

Algorithm 2 *searchForFK* function

```
procedure $SearchForFK(tablePK, columnPK)$
    $result \leftarrow \emptyset$
    for $table \in schema$ do
        for $column \in table$ do
            if $column \neq columnPK$ then
                if $column.type = columnPK.type$
            & $column.length \geq columnPK.length$
            & $column.precision \geq columnPK.precision$
            & $EqualsNames(table, column,$
                                    $tablePK, columnPK)$
             then
                 $result \leftarrow result + (tablePK, columnPK,$
                                    $table, column)$
                end if
            end if
        end for
    end for
    return $result$
end procedure
```

First, the *equalsColumnNames* function checks the length of the column name considered as foreign key candidate (*column*).

- If the length is *equal or greater than 5* characters we check whether the *tablePK* and/or the *columnPK* names is included in the *column* name. We only use the *columnPK* name if its length is greater than 2 characters. Otherwise, we consider that this name is not meaningful enough to be used in the search.

- If the length is *less than 5* characters we do not check if the *tablePK* name is contained in *column* name, mainly because we consider that 5 characters is the minimum length to deal with table names providing meaningful names. So, in this case, we only check the length of the *columnPK* name. If it is greater than 2 characters, the algorithm checks if *columnPK* name is contained in *column* name analysed as foreign key. Otherwise, if the *columnPK* name has a length of 1 or 2 characters, we could suppose that *columnPK*

Algorithm 3 *EqualsNames* function

```
procedure $EqualsNames(table, column, tablePK, columnPK)$
    if $column.name.length \geq 5$ then
        return $(columnPK.name.length > 2$
        $\&\ (column.name.contains(columnPK.name)$
            $or\ column.name.contains(tablePK.name))$
        $)$
        $or$
        $(columnPK.name.length \leq 2$
        $\&\ column.name.contains(tablePK.name))$
    else
        return $((columnPK.name.length > 2$
        $\&\ column.name.contains(columnPK.name))$
        $or$
        $(columnPK.name.length \leq 2$
        $\&\ tablePK.name.contains($
            $column.name.eliminate(columnPK.name)$
                            $.eliminate('\_'))$
        $))$
    end if
end procedure
```

has a name like *'id'* or something similar. In this scenario, a specific check is performed: we eliminate in *column* name the appearances of the *columnPK* name and some other special characters like *'_'*. Then, we verify if the resulting name is part of the *tablePK* name. Let us illustrate this concrete scenario based on an example. We could have a foreign key *column* named *'prid'* which would be analysed in relation to a *columnPK* named *'id'* in a table named *'provider'*. After elimination of the *columnPK* name from *column* name, we would have the string *'pr'* which would be contained in *tablePK* name.

The length thresholds used in the *Schema Analysis* process can be reconfigured. The values chosen in this thesis have been proposed after several tests because they proved to generate the best results in the foreign key detection.

5.2 Data Analysis

The *Data Analysis* process utilises the results generated by the *Schema Analysis* process as starting point. This approach is usually a necessity for large-scale legacy databases, as a brute-force data analysis with respect to detecting all potential foreign keys is usually computationally prohibitive.

Algorithm 4 shows how the data analysis is applied. Taking a set of foreign key candidates, the algorithm calculates the matching of values involved on each candidate. This matching defines how many values in *tableFK.columnFK* can be found in *tablePK.columnPK*. This matching value must be measured in relation to the number of rows in *tableFK* to calculate the percentage of matching values. The number of rows in *tablePK* is reported too for calculation of the matching percentage (i.e. a high percentage value for a *tablePK* containing millions of rows provides stronger support for a hypothetical FK than a high matching percentage of a table with only a few rows).

Algorithm 4 is set up by means of a *threshold* value which is established to only return candidate foreign keys *(tabPK, colPK, tabFK, colFK)* having a percentage value above the threshold value.

Algorithm 4 *Data Analysis* algorithm

```
$result \leftarrow \emptyset$
for $(tabPK.colPK, tabFK.colFK)$
$\in set(tablePK.columnPK, tableFK.columnFK)$ do
    $countPKReg \leftarrow select\ count(*)\ from\ tabPK$
    $countFKReg \leftarrow select\ count(*)\ from\ tabFK$
    $matching \leftarrow select\ colFK\ from\ TabFK$
                  $intersect\ all$
                  $select\ colPK\ from\ TabPK$
    $percentage \leftarrow (matching * 100)/countFKReg$
    if $percentage \geq threshold$ then
        $result \leftarrow result + (tabPK, colPK, tabFK,$
                                  $colFK)$
    end if
end for
return $result$
```

5.3 Static SQL Analysis

Some technologies are able to embed SQL sentences in business logic using delimiter characters. For instance, the JDBC API[1] on the Java platform can embed strings (using double quotes) defining SQL sentences as parameters. Sometimes one must define complete SQL sentences and sometimes it is possible to include a sentence in several fragments of SQL code.

The *Static SQL Analysis* process enables to analyse the source code of the programs in order to identify, parse and exploit the SQL code fragments they contain. It is a linear process composed of 5 steps, where the output of one step constitutes the input of the next step. This process has as first identify the Java files implementing the OSCAR client applications. The main task of the SQL analysis is to parse the client program source code searching for literal strings containing SQL SELECT sentences. Once these sentences have been retrieved, they are parsed to extract the FROM and WHERE clauses. The former is used to relate alias to tables which are required to identify columns involved in a join condition in the latter. For instance, a sentence like *"SELECT * FROM table1 t1, table2 t2 WHERE t1.name = t2.name"* has one join and to identify each prefixed column it is necessary to resolve what table is referenced by each alias. A join condition in a WHERE clause must comply with the following syntax: *'columnA = columnB'*, where columns could be prefixed by table alias or table names. As future work, we have to consider others way to define a join, like using the *in* operator and nested queries. We assume in our analysis process that literal strings composing an SQL sentence are adjacent and properly ordered. We need to make this assumption as we use static analysis and the program code is not interpreted by our analyser. In the following, we explain briefly how the program analysis process is carried out by executing five distinct steps.

- *1st step*: A parser is used to extract string literals, i.e. sequences of characters between two delimiters, in each Java file of the OSCAR system. Delimiters are special language-dependent characters, like the double-quote or the single-quote.

[1]Java Database Connectivity

- *2d step*: Then, we analyse the strings obtained so far and all those which are not related to a SQL SELECT statement are discarded. We assume that a string is only related to a SELECT statement if it does contain some keywords of the SQL SELECT statement syntax or some table and column names occurring in the legacy database schema.

- *3rd step*: Once we have only strings related to SQL SELECT sentences, we are able to re-create a complete SELECT sentence by concatenating two or more strings. To do this, information about the string in needed like: the name of the Java file where it was found, the line number in the file and the position of the first character in the line.

- *4th step*: A second parser is used for extracting the FROM and WHERE clauses from the resulting SELECT sentences. A pre-parsing step deals with discarding literal strings included after the 3rd step but which generates noise due to the presence of parameters that are only determined at runtime, right before the SQL query is executed by the program.

- *5th step*: The final step analyses the content of the WHERE clauses, searching for a join condition and resolving the identity of each column involved in it, by using the table definitions in the FROM clause.

Now, we will use an example to illustrate the SQL analysis implementation. One of the OSCAR Java files is *'OscarCommLocationsDao.java'*, that contains the following code fragment:

Listing 5.1: Code fragment of *OscarCommLocationsDao.java*

```java
@NativeSql({"messagetbl", "oscarcommlocations"})
public List<Object[]> findFormLocationByMesssageId(String messId) {
 String sql = "select ocl.locationDesc, mess.thesubject " +
  " from messagetbl mess, oscarcommlocations ocl " +
  " where mess.sentByLocation = ocl.locationId and mess.messageid = '
     " + messId + "' ";
 Query query = entityManager.createNativeQuery(sql);
 return query.getResultList();
}
```

After applying the first step, the searching of strings found six literals. The pair of numbers indicates the line number and the position of the first character.

Listing 5.2: Output of first step

```
(61,14) "messagetbl"
(61,28) "oscarcommlocations"
(63,15) "select ocl.locationDesc, mess.thesubject "
(64,04) " from messagetbl mess, oscarcommlocations ocl "
(65,04) " where mess.sentByLocation = ocl.locationId and mess.
    messageid = '"
(65,83) "' "
```

The next step excludes the sixth string because it does not contain any SQL keyword nor any table name, nor any column name. The output of this step is:

Listing 5.3: Output of second step

```
(61,14) "messagetbl"
(61,28) "oscarcommlocations"
(63,15) "select ocl.locationDesc, mess.thesubject "
(64,04) " from messagetbl mess, oscarcommlocations ocl "
(65,04) " where mess.sentByLocation = ocl.locationId and mess.
    messageid = '"
```

Our third step creates one SELECT sentence. The two first strings are not included in the sentence, so the output shows three strings:

Listing 5.4: Output of third step

```
(61,14) "messagetbl"
(61,28) "oscarcommlocations"
(63,15) "select ocl.locationDesc, mess.thesubject from messagetbl
    mess, oscarcommlocations ocl where mess.sentByLocation = ocl.
    locationId and mess.messageid = '"
```

The next step discards the two first strings due to their unresolved parameters. Then, it analyses and extracts the FROM and WHERE clauses.

Listing 5.5: Output of fourth step

```
SELECT
FROM :"from messagetbl mess, oscarcommlocations ocl"
WHERE:"where mess.sentByLocation = ocl.locationId
```

```
        and mess.messageid = '"
```

Finally, one join is found and gives rise to a potential foreign key candidate.

Listing 5.6: Output of fifth step

```
/OscarCommLocationsDao.java:
  (PK) "oscarcommlocations.locationId"
  (FK) "messagetbl.sentByLocation"
```

5.4 Hibernate Analysis

A large part of the OSCAR applications uses the Hibernate Object-Relational Mapping (ORM) to access the database. Hibernate allows developers to map Java classes to database tables. Those mappings are usually declared in a mapping file (an XML document) that instructs Hibernate how to map the Java classes to the database tables. Since those files map the relational database schema to the object-oriented structures (Java) of the application program, it can contain valuable information about the database schema and might be an important information source for the detection of RICs. Therefore we consider the Hibernate XML mapping files as another possible way to infer implicit foreign keys.

The Hibernate parser searches in each mapping file for a *'class'* tag, where an entity name is mapped to a table name by means of *'name'* and *'table'* attributes, respectively. If both names are equals, *'table'* attribute could be omitted. In a similar way, the attributes in an entity are declared by a *'property'* tag and *'name'* and *'column'* attributes. Declarations of RICs can be defined using the following tags: *'many-to-one'*, *'one-to-many'* and *'many-to-many'*. Different kinds of RICs are permitted in a mapping file.

5.5 JPA Analysis

JPA[2] is a Java specification for persistence programming which describes the management of relational data in applications. JPA is a generic standard, indepen-

[2]Java Persistence API

dently of any particular ORM middleware. Different concrete ORM middleware products support JPA, including the aforementioned Hibernate middleware. However, using Hibernate “the JPA way” consists in using Java code annotations rather than XML mapping files to specify how persistent objects and their relationships are mapped to relational table structures. The most recent OSCAR components use JPA annotations rather than Hibernate mapping files. Thus, we also consider those JPA annotations as a relevant information source for inferring RICs.

A parser for JPA code annotations was implemented. For each JPA entity file a *'Table'* annotation is searched, where an entity name is mapped to a table name by means of the *'name'* attribute. If both names are equal, the *'Table'* annotation can be omitted. Declarations of RICs are defined using one of the following annotations: *'ManyToOne'*, *'OneToMany'* and *'ManyToMany'*.

5.6 Results

The five techniques described above were implemented for recovering implicit foreign keys. The three first of them were designed and developed by the PhD candidate. In this Section, we will firstly present the results obtained by combining the different techniques and secondly describe *our chosen strategy for accepting and discarding the foreign keys.*

5.6.1 Raw data

After having applied those 5 techniques on the OSCAR system code, we extracted **1.899** foreign key candidates. Figure 5.1 illustrates the distribution through the 5 techniques. **1.818** FK candidates have been detected by the *schema analysis.* **291** FK candidates could have been verified by the *data analysis.* **28** FK candidates have been proposed by the *Hibernate analysis.* **40** FK candidates have been extracted by the *JPA analysis*, while **50** FK candidates have been inferred from the *static SQL analysis.*

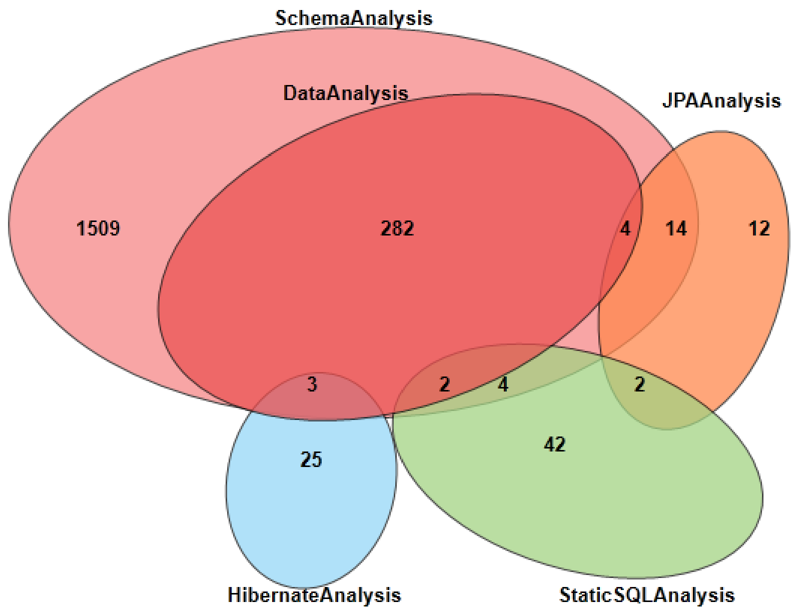


Figure 5.1: Initial report

5.6.2 Acceptance criteria

However another iteration is needed for further exploiting those first results. Indeed, we defined a list of criteria (Table 5.1) allowing us to consider a candidate as *accepted*. Every candidate respecting at least one of those criteria is accepted.
1st criterion. Each FK candidate proposed by the *schema analysis* and having a *matching percentage* above or equal 90% is retained.
2nd criterion. Each FK candidate proposed by the *Hibernate analysis* is retained.
3rd criterion. Each FK candidate proposed by the *JPA analysis* is retained.
4th criterion. Each FK candidate proposed by the *SQL static analysis* is retained only if this FK refers to a *primary/unique key*.

After having applied those four criteria, we moved from **1.899** potential to **215** accepted candidates. Figure 5.2 synthesises these results. The **1.684** remaining ones are considered as *unlikely*. We should point up that: (1) among the 18 candidates in the intersection of *SchemaAnalysis/DataAnalysis* and *JPAAnalysis*, 14 candidates could not be verified by the *data analysis* because the tables were

	Schema	Data	SQL	Hibernate	JPA
1	✔	≥ 90 %			
2				✔	
3					✔
4			PK		

Table 5.1: List of acceptance criteria for the candidate FK.

empty; (2) among the 6 candidates in the intersection of *SchemaAnalysis/Data-Analysis* and *StaticSQLAnalysis*, 4 candidates could not be verified by the *data analysis* for the same reason.

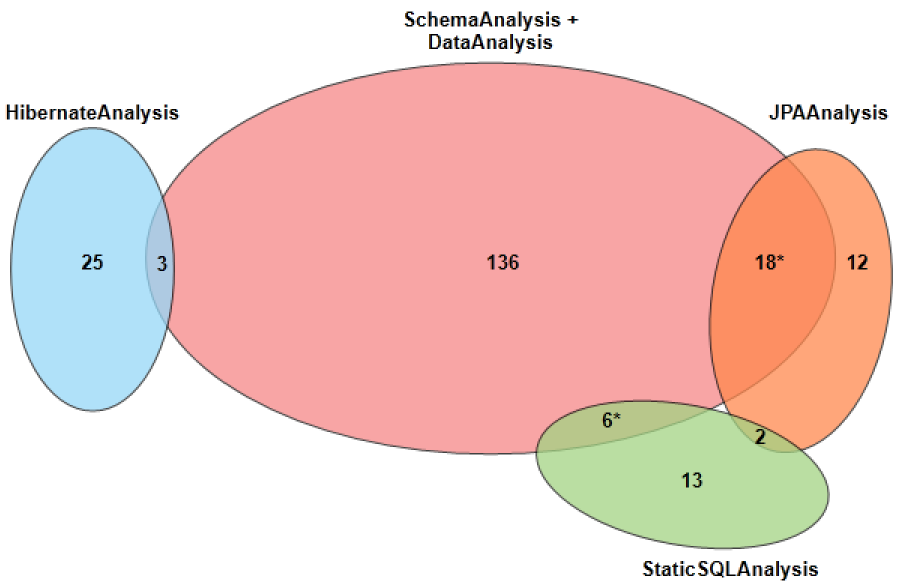


Figure 5.2: Accepted FKs after having applied our acceptance criteria.

5.6.3 Rejected and Unlikely candidates

Up to now, we have defined 2 disjoint categories of candidates, the *accepted* ones and the *unlikely* ones. However, in order to bring more precision to our analysis, we defined a third category, the *rejected* candidates. This is why we identified 4 rejection criteria.

1. **Matching percentage** All the *unlikely* candidates having a *matching value* lower than 90% (*Data analysis*) are rejected.

2. **Bi-directionality** All the *unlikely* candidates such as there exists an *accepted candidate* expressing an opposite direction constraint, are rejected. Figure 5.3 illustrates an example of bi-directionality. The candidate FK **provider**[**provider_no**] → **provider_site**[**provider_no**] is rejected according to the bi-directionality criterion.

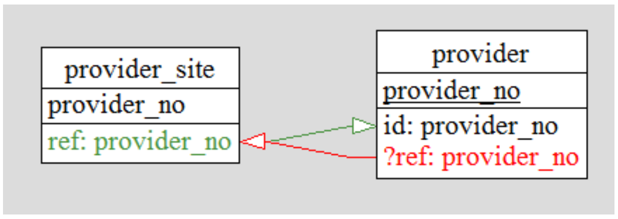


Figure 5.3: Example of bi-directionality.

3. **Unicity** All the *unlikely* candidates such as there exists an *accepted candidate* defined on the same column(s), are rejected. Figure 5.4 shows an example of unicity. The candidate FK **provider_site**[**provider_no**] → **providerbillcenter**[**provider_no**] is rejected according to the unicity criterion.

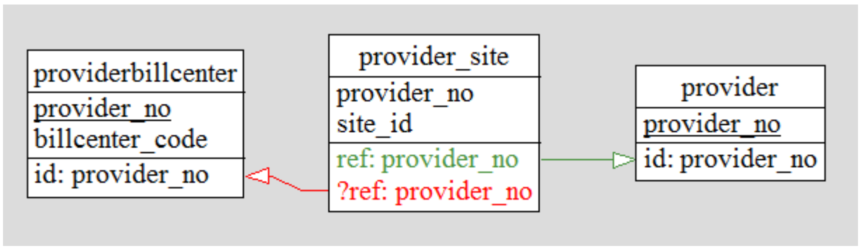


Figure 5.4: Example of unicity.

4. **Transitivity** All the *accepted* candidates which might be derived from other accepted candidates by applying the transitivity property are rejected. An

example of transitivity is depicted by Figure 5.5. The candidate FK **provider-archive**[**provider_no**] $\rightarrow$ **provider**[**provider_no**] is rejected according to the transitivity criterion.

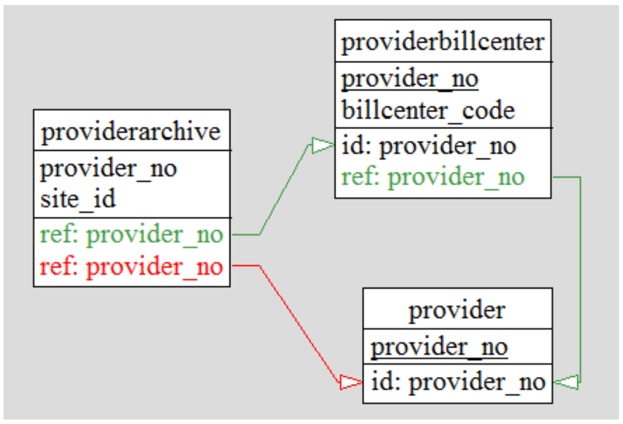


Figure 5.5: Example of transitivity.

Figure 5.6 represents the distribution of the *accepted* candidates through the 5 information sources after having considered our rejection criteria. **146** *unlikely* candidates have been *rejected* because they do not respect the minimal matching value. **1.219** *unlikely* candidates have been *rejected* by **unicity**, **23** *unlikely* candidates by **bi-directionality** while **37** *previously-accepted* candidates have been *rejected* by **transitivity**. It is worth noting that: (1) among the 17 candidates in the intersection of *SchemaAnalysis/DataAnalysis* and *JPAAnalysis*, 14 candidates could not be verified by the *data analysis* because the tables were empty; (2) among the 6 candidates in the intersection of *SchemaAnalysis/DataAnalysis* and *StaticSQLAnalysis*, 4 candidates could not be verified by the *data analysis* for the same reason.

5.6.4 Final results

We presented our automated approach allowing us to recover some implicit foreign keys. Our process is divided into three steps. Firstly, we identified a set of FK candidates. The second step permitted us to *accept* some candidates whereas the third step *rejected* some candidates. Table 5.2 summarises the final results of

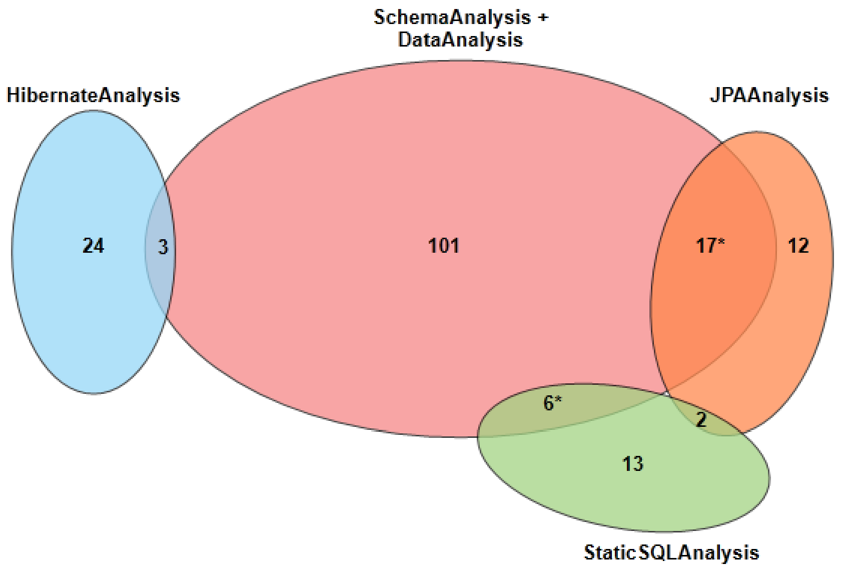


Figure 5.6: Accepted FKs after having applied our rejection criteria.

the third step. **178** candidates are accepted and **1.425** are definitely rejected. However, **296** candidates have not been clearly classified because of the lack of information and would deserve a further study despite their unlikelihood.

Accepted	Unlikely	Rejected
178	296	1.425

Table 5.2: Final results of the discovering FK.

5.7 Discussion and Limitations of the manual approach

The results we obtained manually when identifying FK candidates in the OSCAR system allow us to make some interesting concluding observations, both for RIC detection and foreign key implementation. First, we can observe that the different sources of information have different levels of reliability. While we do not know the actual list of implicit foreign keys that are valid in the OSCAR system, we can already say that the schema analysis technique when used in isolation (without the backup of data analysis) may potentially lead to overly noisy results.

However, there is no perfect source of information that would, alone, be sufficient for identifying the full list of implicit foreign keys of a legacy system. For instance, the Hibernate mapping file and the JPA annotations of OSCAR both are reliable sources of information, and this does not come as a surprise. However, they only allowed to recover a limited subset of the implicit foreign keys in the OSCAR database schema, i.e., those involved in the most recent tables of the database.

The above observation directly relates to the evolution history of the system. We saw, in particular, that the management of implicit RICs in a system may be largely inconsistent, both in the same system version and across the successive versions of the systems. Some old referential constraints have never been explicit declared, some more recent ones have been specified through Hibernate mapping declarations, while some others have been declared through JPA annotations.

Hence, the RIC detection proposal based on the *triangulation* of several RIC identification techniques for confirming/rejecting RIC candidates through cross-checking seems very promising in the context of a legacy system that has already been subject to a long evolution history.

Furthermore, this combination of techniques is also valuable for the FK implementation phase of the reengineering process. Indeed, the results obtained allow one to analyse, for each FK candidate, the impact of making the FK explicit in the database schema. In case the FK candidate is violated by the data: the impact is at least twofold: (1) the inconsistent data will need to be corrected or discarded and (2) source code modifications will be required in order to ensure the adequate management of the FK constraint everywhere the programs insert, update and delete rows in the related tables.

Although it has shown some merits, the multi-source FK identification process suffers from several *limitations*. First, the threshold of 90% for the data consistency heuristics could be further calibrated with respect to the number of rows in the referencing table. In addition, since we did not have access to a ground truth, it was difficult for us to precisely quantify the reliability and the complementarity of the different identification techniques we combine. This will be a prerequisite to further improve our triangulation process and devise a more accurate FK candidate ranking method. Finally, because some OSCAR tables involved in a RIC candidate being empty, our results are partially incomplete as well.

"All that is gold does not glitter,
Not all those who wander are lost."

from The Fellowship of the Ring,
J.R.R. Tolkien
(Suggested by Antonio)

"Some cause happiness wherever they go; others whenever they go."

Oscar Wilde
(Suggested by Veronica)

6 Data Model-Driven Reengineering Process

In this chapter we present an MDE approach for data reengineering. We also analyse to what extent the use of MDE techniques facilitates the implementation of a schema conversion. In particular, we have defined an MDE-based data reengineering approach with which to improve the quality of a legacy schema, which is a common data modernisation scenario.

This chapter is organised as follows. The first section is devoted to briefly describing the improvement of data quality implemented and explaining how a data reengineering process is traditionally tackled. The following section will show a running example and we shall then provide a detailed description of how the three stages of a reengineering process have been implemented in our approach: reverse engineering, restructuring and forward engineering. We shall go on to describe the results of applying our reengineering process to the case study and provide an assessment of the use of MDE techniques in our approach. Finally, we shall present our conclusions.

6.1 Introduction

Legacy databases may have defects as the result of a bad design, limitations of the database platform (e.g. the MyISAM storage engine in MySQL does not support foreign keys), or changes made once the database is in use (e.g. changes made to increase efficiency) [2]. For instance, some of the defects frequently encountered in a database migration are an inappropriate normalisation level or undefined integrity constraints (e.g. foreign keys not declared) [109].

We have therefore addressed the improvement of data quality by fixing defects in the relational data schema and applying data normalisation if necessary. A defect taxonomy can be found in [79], although we illustrate the approach for two cases, namely:

- (i) Undeclared foreign keys. If a value matching is detected between one or more columns in a table and the columns of which the primary key in another table is composed, and the corresponding foreign key does not exist in the source schema, this foreign key will be a candidate to be added to the target schema;

- (ii) Disabled constraints. Foreign keys and check constraints will be candidates to be enabled if they have been disabled for reasons such as adaptation to new requirements or changes made to improve the efficiency.

With regard to normalisation, the legacy schema must be analysed to detect whether the normalisation level is appropriate, and a normalisation algorithm must be performed if the level has to be changed. Figure 6.1 shows the two activities involved in the problem addressed. The task of discovering the two kinds of defects chosen currently involves addressing the well-know problem of recovering implicit information from data: ”eliciting data structures or data properties, such as integrity constraints, that are an integral part of the database, though they have not been explicitly declared in the DDL specifications” [109]. Once the schema has been fixed and correctly normalised, the new schema can be generated (i.e. Data

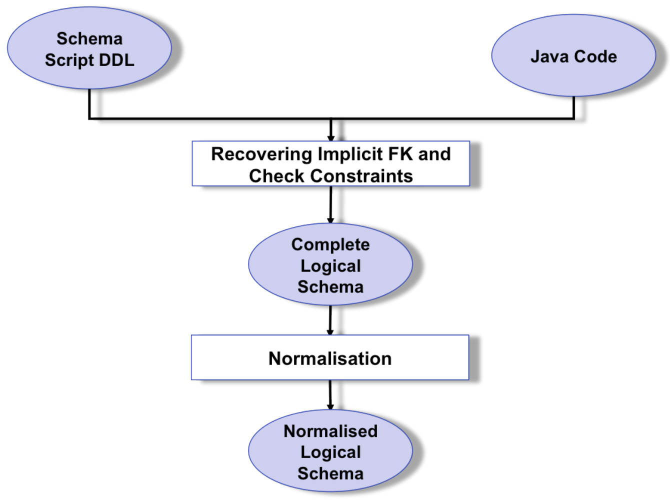

Schema Script DDL
Java Code
Recovering Implicit FK and Check Constraints
Complete Logical Schema
Normalisation
Normalised Logical Schema

Figure 6.1: Problem description

Description Language (DDL) scripts) along with some of the software artefacts required to access data on the target platform (e.g. Java JPA code).

With regard to the three kinds of transformations involved in data reengineering, it is worth noting that in our case (i) the schema conversion has the aim of improving the data quality, (ii) only the data which verifies the constraints included into the new schema are migrated to the new database, and (iii) program conversion is not within the scope of this work.

Data reengineering involves different development tasks such as creating parsers, defining formats with which to represent information, or writing code that implements the required data, schema or program transformations. Table 6.1 shows the tasks to be performed (and artefacts) in the case of the schema modernisation scenario explained above, which has been implemented by means of a software reengineering process. As shown in Table 6.1, extracting knowledge from database requires the creation of the following artefacts: (i) parsers with which analyse schemas and source code, ii) formats in order to represent the information involved into the process, and iii) programs written in purpose-general programming languages (GPL) that can be used to extract knowledge from this information. Data restructuring is performed by manually correcting the defects identified and a tool is then used to detect functional dependencies as a previous step to applying a normalisation algorithm. Tool interoperability is one of the challenges to be tackled in data reengineering [4], which we shall address here by integrating a tool that identifies functional dependencies. Finally, the new schema will be generated by creating the corresponding DDL code and JPA (Java Persistence API) code. Regenerating the database and transforming the existing programs such that they conform to the new schema will require the implementation of the corresponding transformations. ETL (Extract, Transform and Load) processors are used in data conversion, and the programs are normally rewritten so as to access the new schema although other techniques such as the creation of data wrappers could also be applied [17].

Various tools that can be used to facilitate the implementation of data reengineering processes exist. DB-Main [17] is a data engineering environment with built-in DDL parsers for some popular database management systems and analysis program tools, among other features that are useful for data reengineering. program transformations languages such as Stratego [112] can be used to implement the database application transformations in the form of rewritten rules and strategies.

Stage	Tasks (Artefacts)	MDE solution
S1. Reverse engineering	T1.1. Creating Parsers (DDL and Java code fragment) T1.2. Defining formats (DDL, DML, Java Fragments, SELECT sentences, Defects) T1.3. Implementing data and program analysis	- Parsers are automatically created (e.g. using Gra2MoL) - DDL, DML, Fragment, Select and Defects metamodels are created - M2M transformations are created for implementing algorithms
S2. Data Restructuring	T2.1. Correcting defects in the database schema T2.2. Detecting functional dependencies (using specific tool) T2.3. Implementing Normalisation algorithms	- M2M transformation to correct defects - Tool integrating for detecting functional dependencies - M2M transformations to create the normalised schema
S3. Forward Engineering	T3.1. Creating the new schema T3.2. Generating the new database (not addressed) T3.3. Creating the new database access logic (not addressed)	- Model transformation to automatically generate JPA code

Table 6.1: Data reengineering tasks and MDE techniques used in the scenario in which defects are removed.

The reengineering tasks (and artefacts involved) in our data modernisation scenario (”Tasks(Artefacts)” column) and the techniques and tools used in the MDE solution (”MDE Solution” column) are summarised in Table 6.1. After explaining the MDE approach in the following sections, we shall then go on to analyse this table in depth.

The three stages of the applied model-driven data reengineering process are described in detail in next Sections 6.3, 6.4 and 6.5. We then show the results of its application to a real case study (i.e. the OSCAR system) in Section 6.6. Finally, Section 6.7 discusses the assessment carried out through the case study.

6.2 Running Example

In order to guide the next explanations of the three stages, we present first a typical retail application which has to be migrated and whose defects must be removed from the schema. We have considered a simple application which is more illustrative than using the real case study, which will be used before to present the assessment of the approach. Input models for the reengineering process will be obtained from two elements of that application: the logical data schema and the source code of the application. The persistence technology used is JDBC (Java Database Connectivity). The migration process results in a new fixed schema and JPA code with which to access to data by means of an object-relational framework.

Our example schema is a small excerpt which is formed of the following tables: `Customer`, `Order`, `OrderLine` and `Product`:

Listing 6.1: Tables of the running example

```
TABLE Customer (idCustomer, name, city, country)
  PK <idCustomer>
TABLE Order (idOrder, date, idCustomer)
  PK<idOrder>
TABLE OrderLine(idLine, idOrder, amount, idProduct)
  PK <idLine,idOrder>
  FK <idOrder -> Order(idOrder)>
TABLE Product (idProduct, name, price, availability)
  PK <idProduct>
  CK <availability in 'in-stock', 'unavailable',
            'pre-order'> disabled
```

Each table in the example schema includes a primary key (PK); moreover, the `OrderLine` table includes a foreign key (FK) for the `Order` table, and the `Product` table has a check constraint (CK). This schema has a pair of defects or anomalies that could be fixed: (i) the `Order` table contains an attribute that has the same name as the primary key of the `Customer` table (`idCustomer`), although there is no foreign key from `Order` to `Customer`; (ii) the check constraint in the `Product` table is disabled.

A snippet of the example code is shown following. This snippet of Java code merely defines a method which executes a simple query retrieving information concerning orders (its identifier and the customer name). The code that assigns a database connection (i.e. a `Connection` object) to the variable `con` has been omitted for the sake of simplicity. The query results are returned in a `ResultSet` object.

Listing 6.2: Code of the running example

```java
public ResultSet queryOrdersWithCustomer() {
  try {
    Statement statement = con.createStatement();
    System.out.println("Querying orders and their customers.");
    String query = "SELECT o.idOrder, c.name ";
    query = query + "FROM Order o, Customer c ";
    query = query + "WHERE o.idCustomer = c.idCustomer";
    ResultSet results = stm.executeQuery(query);
    return results;
  } catch (SQLException e) {
    System.err.println("SQLException: "
      +e.getMessage());
  }
}
```

6.3 Reverse Engineering Stage: Obtaining the Defect Model

This section shows how MDE techniques could be applied in data reverse engineering for data and code analysis. Before providing a detailed explanation of the model transformation chains that implement the analysis strategies devised (see Figure 6.2), we shall present the Defect metamodel along with the DDL and

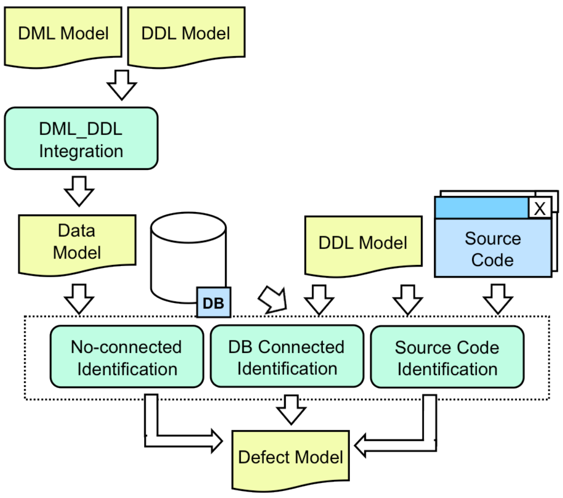


Figure 6.2: Defect identification stage

DML metamodels to which the injected models conform. For each kind of analysis, the strategy applied is first explained. The model transformation chain that implements the strategy is then described in detail; the running example is used to illustrate how the analysis obtains the Defect model in each case. Finally, the strategies are compared, considering several criteria.

6.3.1 Defect Metamodel

As indicated in Section 1.2, we have considered the detection of two defects related to the integrity database: (i) a foreign key is missing, and (ii) either a foreign key or a check constraint is declared but disabled.

Figure 6.3 shows the Defect metamodel that represents the two defects considered. The *Defect* abstract class is the root of the classes that represent defects, i.e. the *ForeignKeyDefect* and *CheckDefect* classes. The former represents the fact that a foreign key has been disabled or does not exist; this class has two references to the *Table* class, signifying that an instance of *ForeignKeyDefect* will have a reference to the table in which a foreign key must be created (*ownerTable* reference) and the table referenced by this foreign key (*referencedTable* reference). The latter represents a check constraint which is disabled; this class has a reference to the *ColumnCK* class which represents a column in which a check constraint is defined by means of the *Expression* class. It is worth noting that this metamodel could

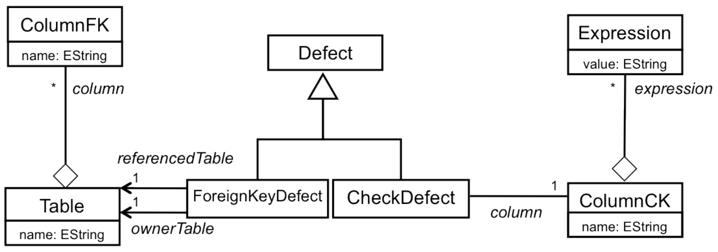


Figure 6.3: Defect metamodel simplified

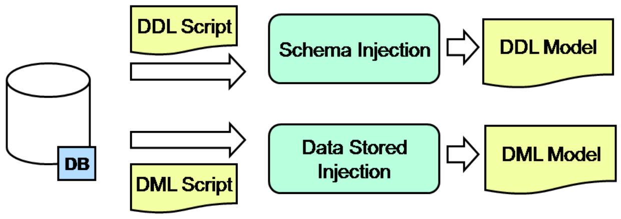


Figure 6.4: Database injection pre-stage

easily be extended to represent other defects, e.g. non-minimal identifiers in a table [79].

6.3.2 Data Model Injection. DDL and DML metamodels

Three models have been injected by using the Gra2MoL language: (i) DDL models represent the logical data schemas and they are obtained from DDL scripts; (ii) DML models represent the data stored in the database and they are obtained from DML scripts; (iii) SQL Code Fragment models, which are obtained from the source code. The first two injections comprise the database injection pre-stage (see Figure 6.4). In order to use Gra2MoL in this stage, we have defined the DDL and DML metamodels and grammars. The third injection will be explained in Section 6.3.4.

Figure 6.5 shows the DDL metamodel. It represents a DDL script as a set of *CreateStatement* which can be of two kinds: *CreateDatabase* or *CreateTable*. A *CreateTable* statement is formed of a set of *ColumnDeclaration* and has references to elements that represent the declarations of the primary key, the foreign key and the check constraints. We have considered only basic logical information and not physical information (e.g., tablespaces) or other elements from database schemas, such as views, indexes, sequences or procedures.

With regard to the DML metamodel, we have solely represented the insert

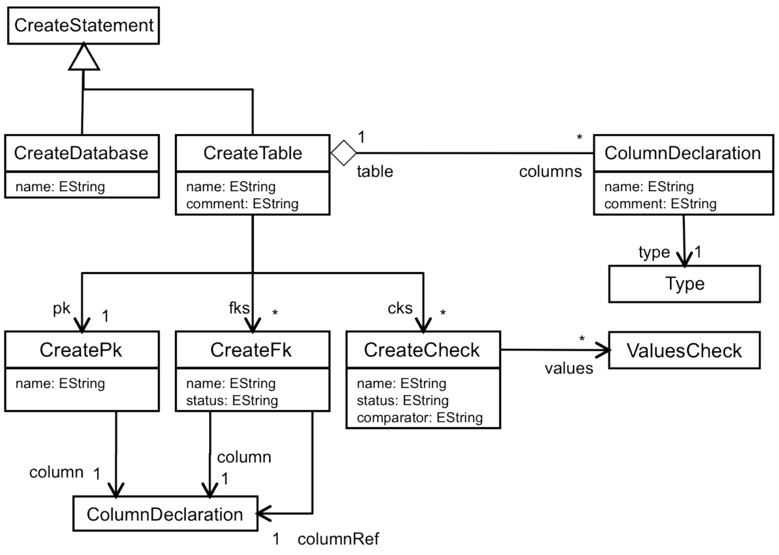


Figure 6.5: DDL Metamodel

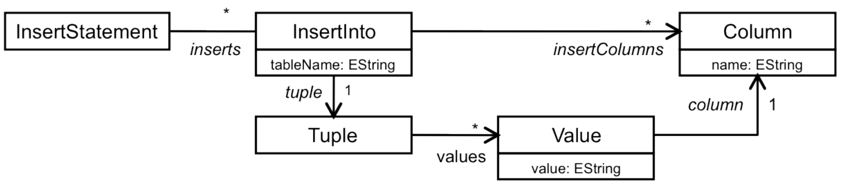


Figure 6.6: DML Metamodel

statements, as shown in Figure 6.6. This statement is used to add tuples to an existing database, so that an instance of the *InsertInto* class has the tuples stored in the legacy database for the referenced table.

6.3.3 Data Analysis

6.3.3.1 Strategy

Once the DML and DDL models have been injected, they are explored in order to discover occurrences of the two kinds of defects considered in this work, and a Defect model is generated. A large number of strategies with which to elicit foreign keys have been proposed over the last two decades. We have applied the algorithm presented in detail in [108] which was devised in the context of this work. Since the focus here is the application of MDE in data reengineering, we shall briefly introduce this algorithm.

The algorithm is organised in two stages. Firstly, all the primary keys are iterated and the set of candidate foreign keys is obtained for each primary key. Given a primary key column, the algorithm searches for other columns in the database schema that could reference it. This search is based on the names and data types of the columns and returns a set of pairs of columns, in which the first column represents a primary key and the second represents a candidate foreign key (composite foreign keys are not considered).

In the second stage, the algorithm iterates over the set of pairs of columns previously obtained and calculates the `percentage` of value matching between the foreign key column and the primary key column for each pair. This percentage is calculated from the total number of tuples in the table containing the potential foreign key (i.e. the percentage indicates how many values in the candidate foreign key column out of the total match a value in the primary key column of the referenced table). When this percentage is higher than a certain `threshold`, the potential foreign key is added to a set which stores the undeclared foreign keys discovered by the algorithm.

6.3.3.2 Implementation

We have experimented with two implementations of the algorithm presented, one of which works with data models (named *DAS-M*, Data Analysis Strategy based on Model) and the other of which directly accesses the database (named *DAS-D*, Data Analysis Strategy with Data stored).

The *DAS-M* strategy does not have to be connected to the database since the analysis is performed on data models injected from DDL and DML scripts. In particular, the algorithm manages a model that integrates the DDL and DML models injected, which is named Data model. A model-to-model transformation generates a Data model from the DML and DDL models. This model contains all the information from both models and it includes a new reference from the values (records) declared in the DML model to columns identified in the DDL model. This integration enables the algorithm to detect the possible defects since data values and their references to columns are easy to use.

Using a Data model instead of directly accessing the database provides independence from the database system and DDL language. However, a significant weakness must be considered owing to the need to mange a large model (i.e. the Data model). The size of this model could be considerably reduced by selecting a representative set of the data stored. This requires an analysis of the data contained in tables before discarding records, since the data values related by means of the foreign keys declared in the schema have to be kept together in the representative set.

Instead of injecting the Data model, the *DAS-D* strategy discovers defects by directly exploring the data stored in the legacy database. The algorithm has been implemented in PL-SQL and accesses the database dictionary for a particular database system (e.g. Oracle database). Therefore, unlike the DAS-M strategy, the DAS-D strategy provides an efficient and complete access to data, but the solution depends on the database system. Writing the algorithm in a procedural language, such as PL-SQL, is easier using a model-to-model transformation language. Most of these languages are designed to declaratively express mappings between models and they support algorithmic strategies, such as those performed in data reverse engineering, by providing imperative constructs that differ for each language.

In order to obtain the Defect model, the PL-SQL program stores the defects discovered in a database and an injection process is then applied. We have taken advantage of the Schemol tool [48], which generates model injectors for relational databases as indicated in Section 2.3. The Schemol program should specify a mapping between the database schema that stores defects and the Defect metamodel. We have therefore defined a database schema in which to store the defects, as shown in Figure 6.7.

6.3.3.3 Example

We shall now show how the defect detection algorithm works for the running example introduced in Section 6.2. We shall suppose the database contains the following tuples for the `Customer`, `Order`, `OrderLine` and `Product` tables.

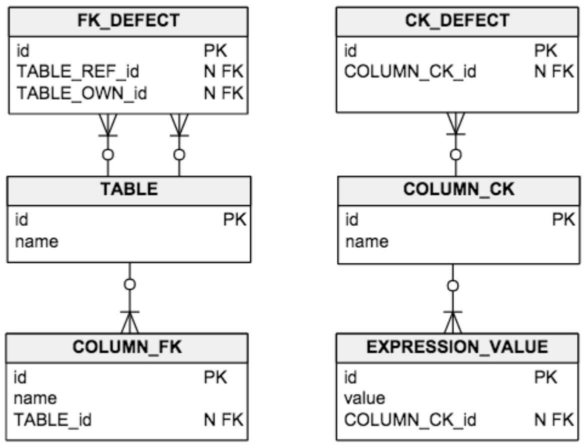


Figure 6.7: Database tables for representing defects

Listing 6.3: Tuples stored in the running example

```
Customer (idCustomer, name, city, country) :
  [LR2, Liam, London, UK],
  [DC3, Dean, Paris, FRA],
  [SA2, Samuel, London, UK],
  [PS5, Peter, London, UK],
  [JM1, James, Paris, FRA]
Order (idOrder, date, idCustomer) :
  [V1, 20/12/2011, LR2],
  [V2, 01/05/2009, JM1],
  [V4, 19/05/2009, DC3],
  [V5, 20/06/2011, unknown],
  [V3, 05/09/2010, LR2],
  [V6, 18/12/2010, DC3]
Product (idProduct, name, price, availability)
  [P1, LCD monitor, 120, unavailable],
  [P2, Printer, 150, in stock],
  [P3, LED monitor, 175, in stock],
  [P4, Hard Disk, 90, pre-order]
OrderLine (idLine, idOrder, amount, idProduct)
  [LV1, V1, 2, P2], [LV2, V1, 1, P4],
  [LV1, V2, 3, P4], [LV2, V2, 1, P1],
  [LV3, V2, 1, P3]
```

The matching operation applied to detect the lack of foreign keys would find that the values of the pair of columns `<Customer(idCustomer), Order(idCustomer)>` match at a percentage of 83%. If this percentage is greater than the threshold value then a new foreign key between both columns would be added to the Defect model (i.e. a *ForeignKeyDefect* instance). This added instance would contain references to the `Order` and `Customer` tables. The reference (`referencedTable`) to `Order` would establish in what table the foreign key would be created (the column of the pair is not primary key), while the reference (`ownerTable`) to `Customer` would establish what table is referenced by the foreign key (the column of the pair is primary key). Furthermore, the matching algorithm would also discover a possible foreign key in the `OrderLine` table (`idProduct` column) for the `Product` table (whose `idProduct` column is primary key), but in this case the percentage calculated would be 100%. Our algorithm would additionally find a check constraint disabled in the `Product` table and calculate that constraint activation would have 100% coincidence in the `Product` table. The activation of this check constraint is therefore considered to be a possible defect and an instance of *CheckDefect* is added to the defect model. The resulting Defect model is shown in Figure 6.8 as a UML object diagram.

6.3.4 Code Analysis

6.3.4.1 Strategy

As indicated above, code analysis is also applied in data reverse engineering. Accessing a database from applications can be performed through the use of various techniques such as embedding SQL sentences in the source code of the business logic using delimiter characters (e.g. JDBC in Java platform) or by means of annotated files (e.g. JPA in Java platform), among others. For instance, the JDBC API on the Java platform can embed strings (using double quotes), defining SQL sentences as parameters. Ot is possible to either define complete SQL sentences or build such sentences using several fragments of SQL code. We have defined an SQL code analysis strategy (named *CAS*, Code Analysis Strategy) which applies text searching pattern-based techniques [113] to the source code of the database applications.This strategy was outlined in [108] and here we shall describe how it

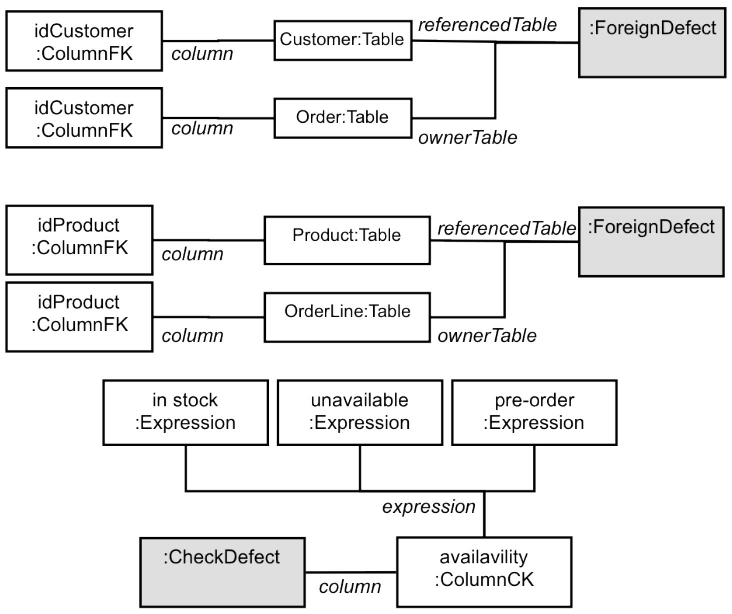


Figure 6.8: Defect model of the running example

has been implemented as part of the chain of model transformations. The process of source code analysis consists of five steps in which the output of one step constitutes the input of the next one.

- *Step 1*: Client program source code is parsed in order to search for string literals, i.e. sequences of characters between two delimiters (e.g. the double-quote or the single-quote). String literals are keep orderly according to how they are found in the program code.

- *Step 2*: The strings retrieved are analysed in order to select SQL SELECT statements, and those strings that are not selected are discarded. We assume that a string is only related to a SELECT statement if it does contain some keywords of the SQL SELECT statement syntax or some table and column names that occurr in the legacy database schema.

- *Step 3*: Once we have obtained only strings related to SQL SELECT sentences, we are able to re-create in a file a complete SELECT sentence by concatenating two or more strings. To do this, information about the string is needed, such as the name of the code file in which it was found, the line number in the file and the position of the first character in the line.

- *Step 4*: A second parsing is applied to the resulting SQL SELECT sentences in order to extract the FROM and WHERE clauses. A pre-parsing step deals with discarding literal strings included after the third step but which generate noise owing to the presence of parameters that are only determined at runtime, just before the SQL query is executed by the program.

- *Step 5*: The final step analyses the content of the WHERE clauses, searching for a join condition and resolving the identity of each column involved in it, by using the table definitions in the FROM clause. A FROM clause is used to relate an alias to tables which are required to identify columns involved in a join condition in the WHERE clause. For instance, a sentence like *"SELECT * FROM table1 t1, table2 t2 WHERE t1.name = t2.name"* has one join, and in order to identify each prefixed column it is necessary to resolve what table is referenced by each alias. A join condition in a WHERE clause must comply with the following syntax: *'columnA = columnB'*, where columns could be prefixed by table alias or table names.

Our approach for the static analysis of Java code would also be applicable to technologies which use SQL code to directly (complete or fragmented) access data (e.g., Oracle Forms code). In this case, the first stage, whose aim is to extract SQL sentences, would not be necessary. It is worth noting that the strategy implemented is only valid for technologies which have SQL sentences embedded in them. Others that use different techniques to access database (e.g., annotations on JPA) could not be used in our approach. Moreover, as future work, we should consider others ways in which to define a join, such as using the *in* operator and nested queries. In our analysis process we assume that the literal strings comprising an SQL sentence are adjacent and properly ordered. It is necessary to make this assumption as we use static analysis and the program code is not interpreted by our analyser.

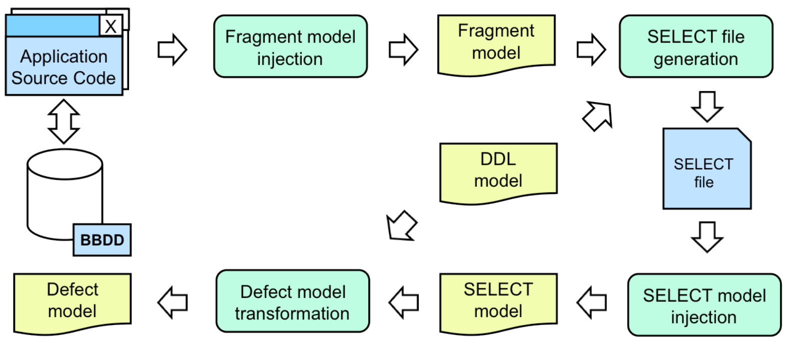


Figure 6.9: Defect identification based on analysing of the application source code

6.3.4.2 Implementation

A detailed explanation of the model transformation chain defined to implement the five-step strategy of SQL code analysis, which is outlined in Figure 6.9, is provided as follows. The code shown in Section 6.2, which access the data schema in the running example, will be used to illustrate each of the stages. Henceforth we shall use the *code fragment* term to refer to String literals in JDBC code and *SQL fragment* to refer to those code fragments that contain part of an SQL sentence. Whatever the technology applied, SQL code fragments must be identified and parsed in order to perform an analysis whose aim is to discover possible defects in the database schema.

Fragment Metamodel

The source code is represented as models that conforms to the Fragment metamodel, which is shown in Figure 6.10. This metamodel is very simple and represents a source code file (*File* class) as a set of code fragments (*Fragment* class); a *File* has a name and a *Fragment* contains information on a code fragment: the number of the line in the file, the position in which the fragment starts in the line, and the fragment text. There are two references from *File* to *Fragment*: (a) *fragments* is used to refer to each fragment found in source code (please recall that in this strategy a fragment is a char sequence enclosed by some delimiter, in our case, a double quotes char); (b) *SELECTfragments* is used to refer to each

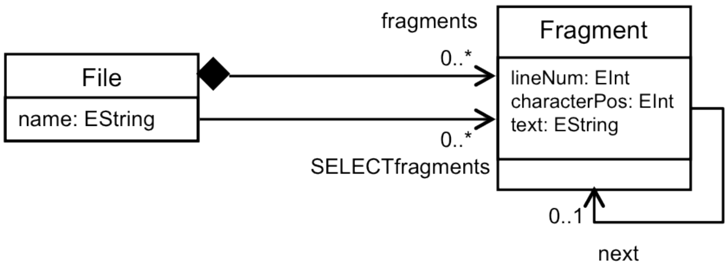


Figure 6.10: Fragment Metamodel

fragment that contains the keyword "SELECT" to identify the beginning of an SQL query (this reference will be populated in the following steps of the process).

Each stage of the transformation chain defined is described below. The corresponding step of our strategy is indicated for each stage.

Step 1. Fragment model injection

A Fragment model is injected from the source code of the legacy application. This injection task has been implemented with the Gra2MoL language introduced in Section 2.3 and the grammar required only defines the main clauses in a SELECT sentence: *select*, *from*, *where*, *group by*, *having* and *order*. It is necessary to identify each table (with its alias) involved in a SELECT query by using the *from* clause, while the how columns are related to each other by means the *where* clause. The rest of the information specified in the SELECT clause is not necessary for the analysis. Note that this task, in addition to injecting the input model into the transformation chain, performs the parsing indicated in step 1 of our strategy.

The Fragment model generated for the running example code would contain five Fragment elements representing the following code fragments.

Listing 6.4: Fragment elements after Step 1.

```
(1) - "Querying orders and their customers."
(2) - "SELECT o.idOrder, c.name "
(3) - "FROM Order o, Customer c "
(4) - "WHERE o.idCustomer = c.idCustomer"
(5) - "SQLException: "
```

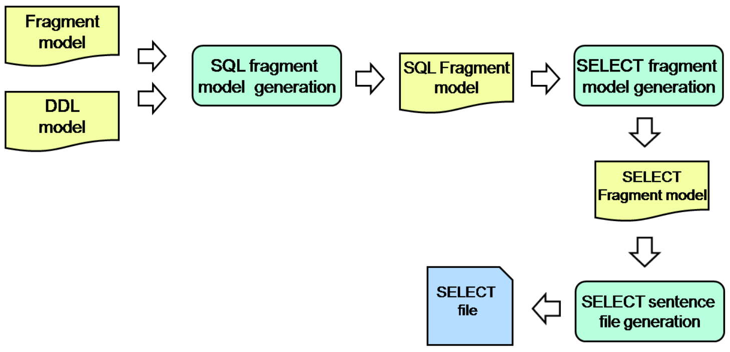

Fragment model
DDL model
SQL fragment model generation
SQL Fragment model
SELECT fragment model generation
SELECT Fragment model
SELECT sentence file generation
SELECT file

Figure 6.11: SELECT file generation.

Steps 2 and 3. SELECT file generation

The goal of this second stage is to create a file that will store all the SELECT SQL sentences in the source code analysed. This task is accomplished by means of a three-step model transformation chain as shown in Figure 6.11. This transformation chain performs a simple parsing on the original code in order to find the SELECT sentences and has two input models: the fragment model injected in the previous stage and the DDL model.

Firstly, all the fragments that are not SQL fragments are removed from the Fragment model. To achieve this, each fragment is analysed to check whether the text contains any keywords of the SQL grammar. In our example, the first and fifth of the fragments shown above would be removed (*"Querying orders and their customers."*, *"SQLException: "*). Another M2M transformation is then applied to the Fragment model obtained in order to remove those fragments that do not represent SELECT SQL sentences. In the example, all the fragments correspond to SELECT SQL sentences. Finally, the new Fragment model is the input to an M2T transformation that generates a textual file containing the SQL code of the SELECT sentences in the legacy code. We shall refer to this file as the SELECT file. In the case of the example code, the SELECT file would contain the following lines:

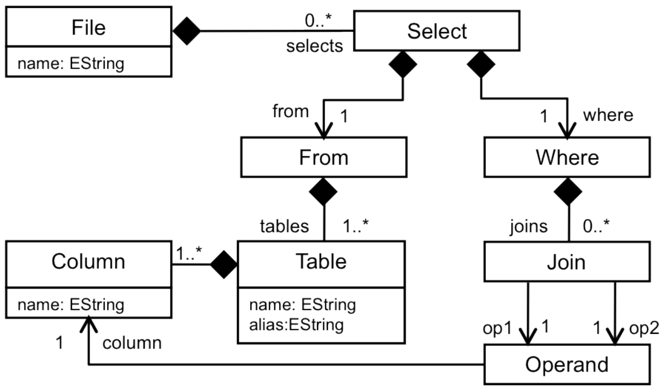


Figure 6.12: SELECT metamodel

Listing 6.5: Fragment elements after Step 3.

```
(2) - "SELECT o.idOrder, c.name "
(3) - "FROM Order o, Customer c "
(4) - "WHERE o.idCustomer = c.idCustomer"
```

Step 4. SELECT model injection

In this third stage, Gra2MoL is again used, now to inject a SELECT model from the SELECT file generated in the previous stage. This model conforms to the SELECT metamodel shown in Figure 6.12, which represents the clauses of a SELECT sentence. The SELECT model could in fact have been generated in the previous stage rather than generating a text file. How-ever, the M2M transformation required would involve a parsing of fragments, which is not necessary when Gra2MoL is used. Please recall that Gra2MoL automatically generates a parser for the input grammar of the transformation. Writing the M2T trans-formation that generates the SELECT file and the T2M trans-formation that injects the SELECT model is simpler than writing the M2M transformation that converts a Fragment model into a SELECT model, and less effort is therefore required.

A significant advantage of generating the SELECT file is that the SQL sentence discovering process is clearly separated from the other stages in which the SELECT sentences obtained are analysed. This makes it possible to perform or not perform the three first steps, de-pending on the technology to be analysed. For example, in

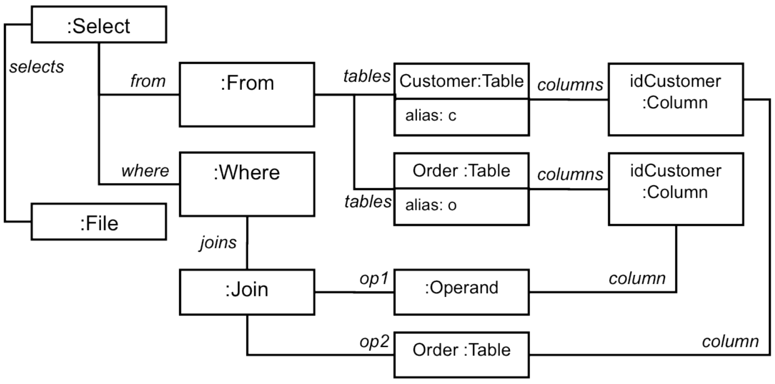


Figure 6.13: SELECT model

the case of the JDBC technology, the generation of the SELECT file is necessary as the SELECT sentences are embedded in Java code. However, this step would not be necessary in the case of Oracle Forms.

The SELECT model generated for our example code is shown in Figure 6.13.

Step 5. Defect model generation

In the last stage, the SELECT model is analysed to find possible defects in the legacy data schema, which are output in the form of a Defect model. This task is accomplished by means of an M2M transformation which has two input models: the SELECT and DDL models. Each SELECT element is analysed to check the two defects considered in our work, namely undefined foreign keys and disabled check constraints. SELECT statements are analysed. The DDL model is consulted whenever a possible defect is found. The analysis is performed as follows. WHERE clauses are examined to check whether they contain either joins or a constraint on the values of a column. When a join is found, the DDL model is accessed to check whether one of the two columns is primary key. Three cases are therefore possible:

- None of the columns is primary key and no *ForeignKeyDefect* is therefore created.
- One of the columns is primary key. The DDL model is therefore accessed in order to discover whether the schema includes the foreign key. A For-

eignKeyDefect is created if such a foreign key does not exist previously in the DDL model.

- Both of the columns are primary key. The DDL model is then accessed in order to discover whether the schema includes one foreign key for one of the columns. If there is one, then no ForeignKeyDefect is created, because the only possible foreign key is already in the DDL model. Otherwise two *ForeignKeyDefect* (one for each column) are created. However, since a schema can not include a foreign key from a column *a1* to a column *b1* if a foreign key from *b1* to *a1* exists, it would be mandatory to discard one of the two *ForeignKeyDefect* created (a wizard is provided to accept or reject each foreign key proposed).

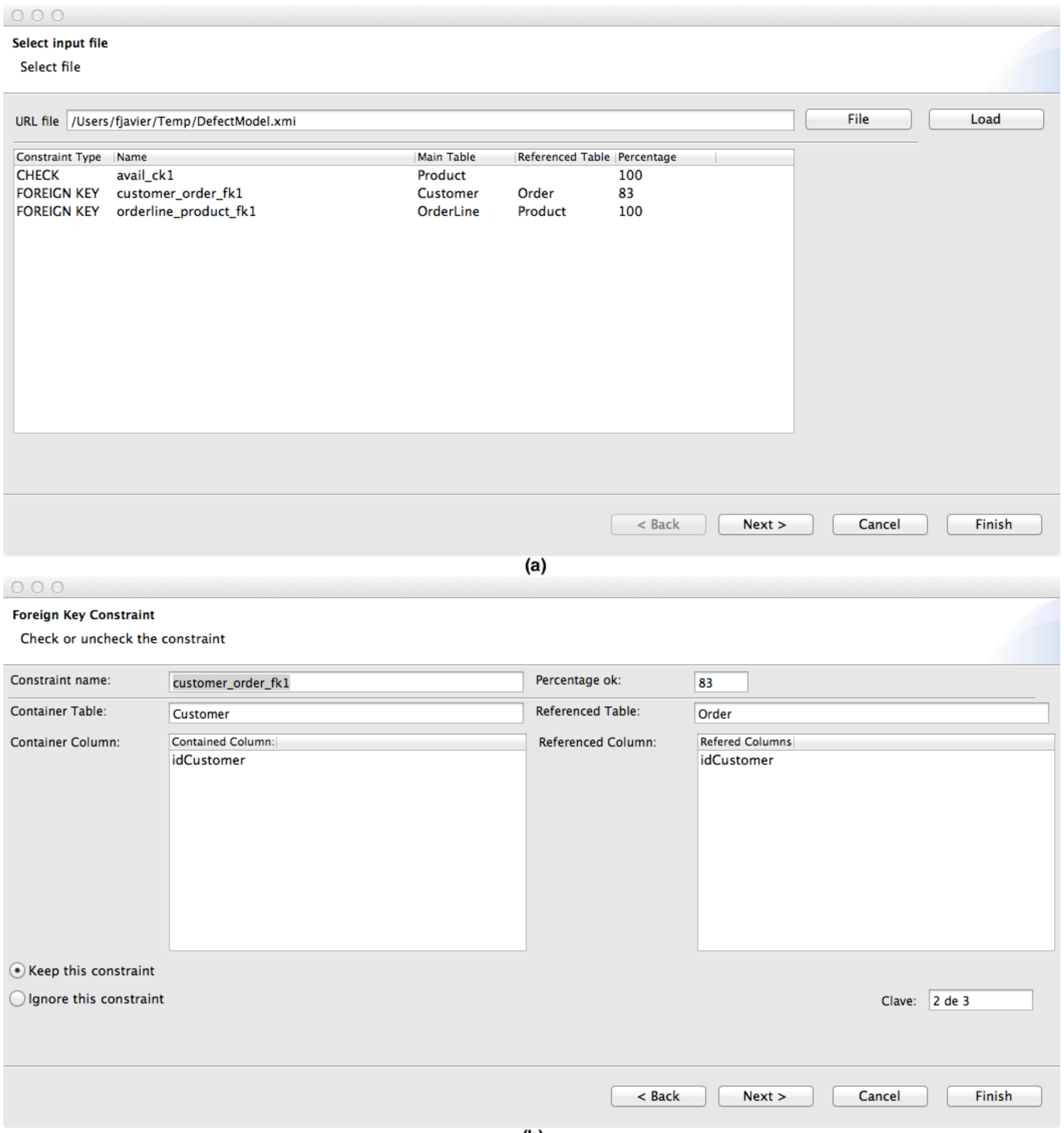
When this strategy is applied to our code example it detects a possible foreign key between the columns `idCustomer`(`Order`) and `idCustomer`(`Customer`) since `idCustomer` is a primary key in `Customer` and `idCustomer` is not declared as a foreign key in `Order`. The Defect model obtained is composed solely of the first ForeignKeyDefect shown in Figure 6.8.

It should be noted that the Defect model includes *possible* defects in the legacy data schema, but that these may originate in from design choices made by the database administrator. We have therefore developed a wizard (see Figure 6.14) in order to allow the administrator or reengineering team leader to confirm what defects should really be fixed. The wizard shows the defects in the Defect model and allows the user to select those to be removed, thus generating the final Defect model. In Figure 6.14a the wizard shows all the defects found during the analysis. By selecting each defect, the wizard shows information related to the constraint proposed. Figure 6.14b shows information about a foreign key.

The wizard allows us to *keep* or *ignore* the constraints proposed to deal with each defect. When the constraint is kept, a boolean attribute for the *Defect* class in the Defect model is set to *true*.

6.3.5 Comparison of Strategies

To conclude this section, the three strategies presented are compared according to five criteria: technology (database system and language) independence, database



Select input file

Select file

URL file: /Users/fjavier/Temp/DefectModel.xmi

Constraint Type	Name	Main Table	Referenced Table	Percentage
CHECK	avail_ck1	Product		100
FOREIGN KEY	customer_order_fk1	Customer	Order	83
FOREIGN KEY	orderline_product_fk1	OrderLine	Product	100

(a)

Foreign Key Constraint

Check or uncheck the constraint

Constraint name: customer_order_fk1

Percentage ok: 83

Container Table: Customer

Referenced Table: Order

Container Column: Contained Column: idCustomer

Referenced Column: Refered Columns: idCustomer

[✓ Keep this constraint]

Ignore this constraint

Clave: 2 de 3

(b)

Figure 6.14: Defects in the wizard

size independence, reliability and efficiency.

- *Database system independence.* The data analysis strategy that uses a Data model (DAS-M) and the code analysis strategy (CAS) are completely independent of the database system. Code fragments analysed in the CAS strategy are database independent, whereas independence is achieved by injecting DML and DDL models in the case of the DAS-M strategy. However, the data analysis strategy that directly queries the database (DAS-D) is database dependent because the code that explores the data schema performs queries using the data dictionary.

- *Data access technology independence.* Unlike the CAS strategy, DAS strategies are independent of languages or technologies used to implement client applications because they deal only with tuples stored in a database and not with any source code.

- *Database size independence.* The CAS strategy is independent of database size since it requires only the source code of client applications. However the DAS strategies explore the legacy database signifying that the database size will negatively affect the execution time. Moreover, the size of a DML model may become so large that its manipulation requires model repositories, such as CDO [114] or Morsa [115].

- *Reliability of the detection.* The CAS strategy is more reliable than DAS strategies since defects are detected by analysing the SQL code which reflects how the schema is really used to access data (i.e. the usage of the database schema by the programers), rather than applying a set of heuristics to the data values, e.g. the matching between columns applied to detect foreign keys.

- *Efficiency.* DAS strategies are based on exploring data values and schema constructs whereas the CAS strategy analyses source code in the search for data access sentences. In both cases, efficiency will depend on certain factors such as size of programs or databases. However, the CAS strategy searches directly for data access sentences which contain information regarding possible defects, whereas the DAS strategies first search in the schema structures, after which a data stored matching is necessary. This matching uses heavy queries crossing records between columns which are less efficient than directly extracting knowledge from the data access sentences found in the CAS strategy.

The comparison for the three strategies is summarised in Table 6.2. In the first three rows of the table, the mark indicates what strategies satisfy the corresponding criterion of independence, while the more reliable and efficient strategy is marked in the last two rows.

	DAS-D	DAS-M	CAS
Database System Independence		✔	✔
Data access technology Independence	✔	✔	
Database Size Independence	✔		✔
Reliability			✔
Efficiency			✔

Table 6.2: Comparison of strategies implemented for the FK discovering.

6.4 Restructuring Stage: Applying Defect Correction and Normalisation

In this section we shall describe how the data restructuring stage has been carried out. As shown in Figure 6.15, the defects contained in the Defect model are first removed in order to improve the quality of the legacy schema. The new Data model is then analysed to check whether or not the normalisation level of the schema is appropriate.

6.4.1 Defect correction

The defect correction task is implemented by means of an M2M transformation whose inputs are the Defect model and the Data model and which generates a

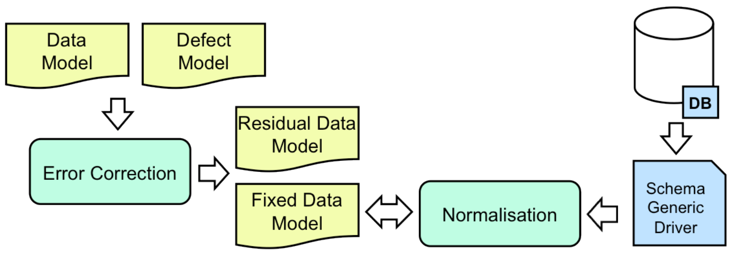


Figure 6.15: Defect correction stage

new fixed Data model. This transformation iterates over the defects in the Defect model and for each Defect instance modifies the Data model according to the kind of defect. For instance, in the case of a *ForeignKeyDefect* instance, a new foreign key must be added to the Data model. This involves the creation of a *CreateFK* instance whose *column* and *ColumnRef* references must be linked to the *ColumnDeclaration* specified in the Defect model. Note that the Defect model used in this stage is the model generated by interacting the user with the defect selection wizard.

Two Data models will therefore be managed from this stage on:

- The first is a Data model (*fixed Data model*) which contains the new constraints added after the Defect correction step and the data values which verify these new constraints.
- The second is a Data model (*residual Data model*) which maintains the original schema and contains only those data values that do not verify the new constraints added to the previous Data model. This residual Data model allows database administrator to decide what to do with the excluded data values which do not conform to the new constraints added to the fixed Data model.

6.4.2 Normalisation

Database normalisation is a two-fold process consisting of functional dependency identification and normalisation. First, functional dependencies [40] are identified in order to detect database redundancies and inconsistencies. Next, a strategy based on decomposition or synthesis algorithms is applied in order to eliminate

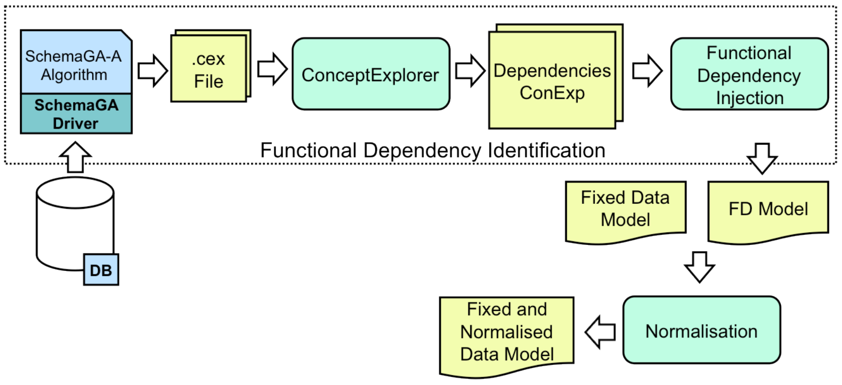


Figure 6.16: Normalisation task

functional dependencies. Figure 6.16 shows the model transformation chain devised to implement this process, which is explained below.

Identification of functional dependency .

As we commented in Chapter 2, the functional dependency theory is used in relational database design, and particularly, in database normalisation [40]. A functional dependency is a semantic constraint between two sets of columns in a table. Given a table T, a set of columns Y in T is said to be functionally dependent (denoted X ->Y) on a set of attributes X, also in T, if and only if for each X value (i.e. a given set of values for each attribute in X) that appears in T the corresponding Y value is unique. The Y set is known as the *dependant* part and the X set as the *determinant* part. In the running example, a dependency: `city ->country` can be found in the *Customer* table.

In order to represent functional dependencies as models, we have defined the FD metamodel which is shown in Figure 6.17. The main concepts of a relational schema are represented in this metamodel, such as *Table*, *Column* and *Restrictions* (primary key and foreign key). It also includes

some elements with which to represent specific information on functional dependencies. *FunctionalDependency* represents a functional dependency and has references to the dependant and determinant parts (i.e. a set of *Columns*); the *RestrictionColumn* represents those columns involved in a constraint that are related to another *RestrictionColumn* (by means of the *pkRestrictionCol* reflexive reference), i.e. columns in a foreign key which have to refer to the column in a primary

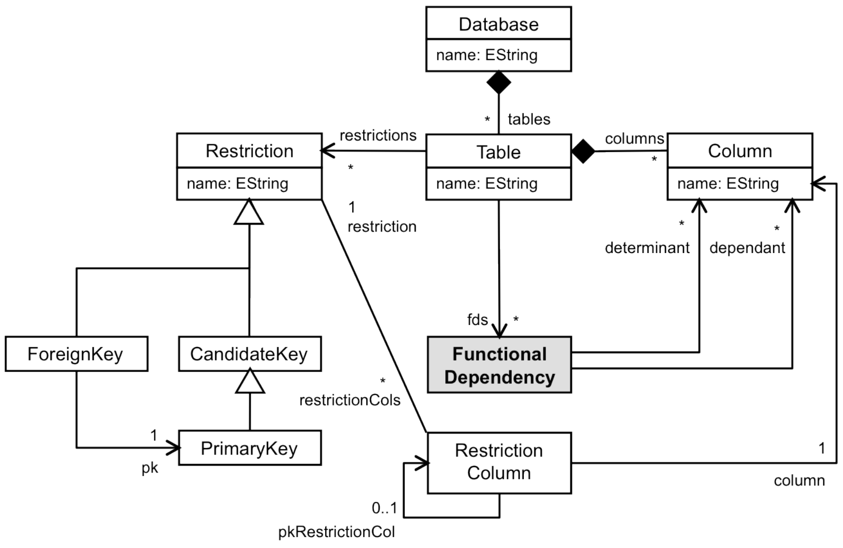


Figure 6.17: Functional Dependencies metamodel

key to which they are related (reference *column*). A *Restriction* is composed of a collection of *RestrictionColumn* (by means of the *restrictionCols* reference).

An excerpt of the FD model for the running example is shown in Figure 6.18. This model includes a *FunctionalDependency* that represents the functional dependency `city->country` in the `Customer` table, and three *RestrictionColumns* that represent: the following restrictions: the `idCustomer` column is PK in the `Customer` table, the `idOrder` column is PK in the `Order` table, and the `idCustomer` column is a FK in the `Order` table. The model also shows a *pkRestrictionCol* reference between two *RestrictionColumns* that relates the column in the FK to the column in the PK.

The identification of functional dependencies has been performed by using the *Formal Concept Analysis* (FCA) method. The fundamental notions of FCA are *formal concept*, *concept lattice* and *formal context.*

A formal context is defined by using three elements: a set of objects O (cluster of objects), a set of attributes or properties P (cluster of attributes), and a binary relation R between O and P. A pair $< o, p >$ ($o \in O$ and $p \in P$) belongs to R if the object o satisfies the property p. A formal concept can be represented using a *Boolean* matrix whose values indicate whether or not an object $o \in O$ satisfies a property $p \in P$. This matrix has a row for each object in the set O and a

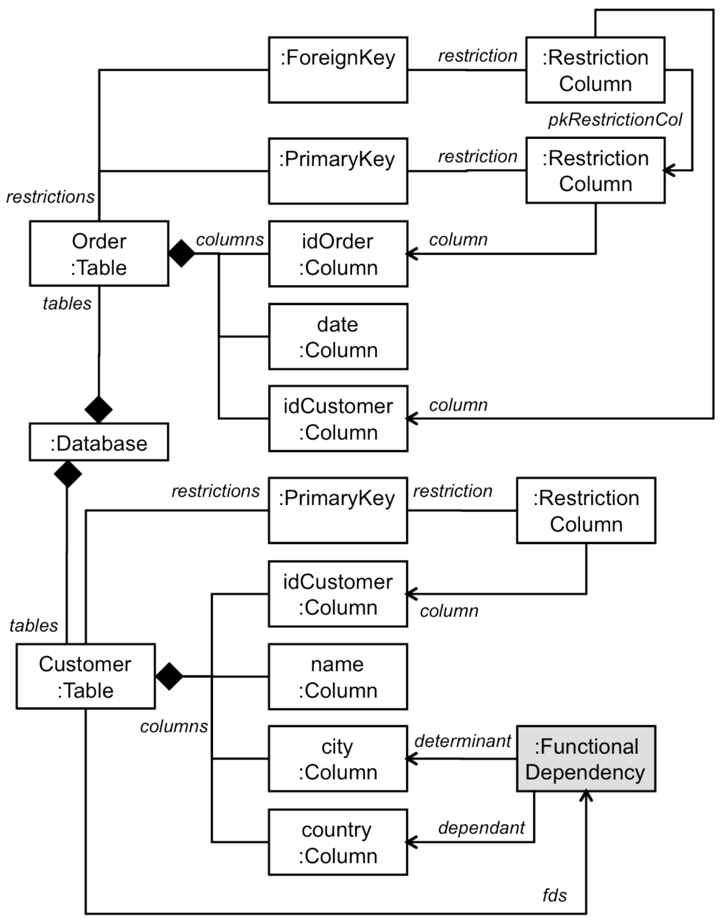


Figure 6.18: Functional Dependencies model for the running example

column for each property in the set P. Given a formal concept, a pair $< X, Y >$ ($X \subseteq O$ and $Y \subseteq P$) is said to be a formal context if and only if each of the objects in the set X satisfies all the properties in the set Y, and Y includes only those properties satisfied by all the objects in X. Finally, the set containing the all formal contexts defined for a given formal concept is called a concept lattice. Data dependencies can be extracted from a concept lattice by obtaining a set of property implications. A *property implication* is an expression $A \Rightarrow B$ where A and B are a set of attributes ($A \subseteq P$ and $B \subseteq P$), signifying that *if an object has all the properties in A then it also has all the properties in B.*

When FCA is applied in order to detect functional dependencies in relational databases [32], a formal context is defined for each table. In these contexts, the cluster of properties or attributes is formed of the columns in the table and the cluster of objects is formed of all the possible combinations of two tuples which are formed of the set of tuples stored in the table. In the boolean matrix defined for each table, each value indicates whether or not the two tuples associated with the row have the same value as the attribute associated with the column. Functional dependencies are therefore expressed as attribute implications.

Given the following tuples for the `Customer` table used in the the running example, the formal context is given by the matrix shown in Table 6.3, in which the `idCustomer` column is represented by (`a`), `name` column by (`b`), `city` column by (`c`) and `country` column by (`d`). With regard to the implications, since there are four pairs of tuples in which the *city* attribute value determines the *country* attribute value, this denotes a functional dependency for the implication *city* $\Rightarrow$ *country*.

Listing 6.6: Tuples of the *Customer* table.

```
Customer (idCustomer, name, city, country) :
   [LR2, Liam, London, UK],
   [DC3, Dean, Paris, FRA],
   [SA2, Samuel, London, UK],
   [PS5, Peter, London, UK],
   [JM1, James, Paris, FRA]
```

There are several software tools which allow a FCA-based data analysis to be applied. We have used a well-known FCA tool named *Concept Explorer* (ConExp) [31], but other FCA tools could also be used. ConExp requires as input a

		a	b	c	d
[LR2, Liam, London, UK]	[DC3, Dean, Paris, FRA]				
[LR2, Liam, London, UK]	[SA2, Samuel, London, UK]			✘	✘
[LR2, Liam, London, UK]	[PS5, Peter, London, UK]			✘	✘
[LR2, Liam, London, UK]	[JM1, James, Paris, FRA]				
[DC3, Dean, Paris, FRA]	[SA2, Samuel, London, UK]				
[DC3, Dean, Paris, FRA]	[PS5, Peter, London, UK]				
[DC3, Dean, Paris, FRA]	[JM1, James, Paris, FRA]			✘	✘
[SA2, Samuel, London, UK]	[PS5, Peter, London, UK]			✘	✘
[SA2, Samuel, London, UK]	[JM1, James, Paris, FRA]				
[PS5, Peter, London, UK]	[JM1, James, Paris, FRA]				

Table 6.3: Formal Context for the running example.

formal context which is provided as an XML file. A formal concept analysis is performed to obtain a set of implications. Each line of the output has the following structure in BNF notation: `number <number of objects> <implication>`, where `number` denotes a unique numeric identifier, `number of objects` denotes the number of objects which verifies the `implication` that is expressed in the form `premise ==> conclusion`, where `premise` and conclusion each denote a set of attributes, which express the determinant and dependant parts of the concept analysed, respectively. For the `Customer` table example, ConExp would generate a file which would only contain the following line: `1 <4> city ==> country`.

After applying ConExp tool (see Figure 6.16), its output file (i.e. functional dependencies) is injected into a functional dependency model that conforms to the FD metamodel shown in Figure 6.17. This injection has been also implemented by Gra2MoL and we have defined a simple grammar for the format used in ConExp in order to express implications.

Automating the usage of ConExp

When using ConExp to detect functional dependencies, the user has to input the context matrix for each table and the tool then generates a file containing the implications. The task of creating the XML documents (.cex files) that define the context matrices is tedious, and we have therefore developed a *Schema*

Generic Access (SchemaGA) driver which provides a uniform accessing mechanism for database schemas and thus allows the definition of a *Schema Generic Access Algorithm* (SchemaGA-A) with which to explore any database schema, as shown in Figure 6.16. Given a particular database system (e.g. Oracle or MySQL), the concept matrices for any database schema could therefore be automatically generated by using the SchemaGA-A algorithm. SchemaGA has been implemented using JDBC/Java and it has taken advantage of the mechanisms provided by this technology to access the database metadata. The driver has an API which allows a database schema to be explored in order to obtain information about its elements, such as tables, attributes (columns) and rows (tuples). The following code fragment illustrates how the driver could be used:

Listing 6.7: Example of the use of the driver.

```java
Schema schema = SchemaConnection.getSchema(
    SchemaConnection.JDBC, driverName,
    urlConnection, user, password);
Table table=schema.getTable(tableName);
HashMap<String, Attribute> attributes=table.getAttributes();
Set<String> attributesNames=attributes.keySet();
```

The `schema` variable refers a `Schema` object that contains information about the database schema. A `Schema` object is obtained by means of the `getSchema()` class method in the `SchemaConnection` class. This method has five parameters, which are the type of the database connection and four strings that express: the class name of the connection driver, the url of the database connection, and the user and password needed to access the database. The `Schema` class has the `getTable()` and `getTables()` methods which allow one or all the tables of the database to be obtained, respectively. The former is used in the code example to obtain the table whose name is provided by the `tableName` variable. In a similar way, rows and attributes can be obtained from a `Table` object by means of the `getAttributes()` and `getRows()` methods, respectively.

Moreover, in the case of a large database schema, creating all the context matrices could be a time-consuming operation since it is necessary to generate all the combinations of pair of tuples (2-combination) for each table. We have improved the efficiency, by implementing the inverted indexes strategy [32] which can reduce the pairs of tuples that must be combined. It only iterates once for each row of a table, and pairs of tuples without a matching attribute are omitted.

6.4.3 Schema normalisation

Once the functional dependencies have been obtained in the FD model, a normalisation process is applied if needed, which has also been implemented as a two-step model transformation chain. The fixed Data model is first transformed into a model in the first normal form (1NF), which in turn, is transformed into a model in the third normal form (3NF) or in the Boyce-Codd normal form (BCNF) by means of a second M2M transformation. The first M2M transformation implements a simple algorithm which extracts multi-valued columns from a new table, and the second implements both the decomposition and synthesis methods [40].

As demonstrated in database design theory, the decomposition method can generate a final normalised schema whose structure is more similar to the original schema than when using the synthesis method. We have chosen to attempt firstly a decomposition, in which the synthesis is performed only if the decomposition cannot remove all the functional dependencies from the tables. Given the FD model obtained for the running example in the previous stage of functional dependency identification, which would include the `city ->country` functional dependency for the `Customer` table, the normalisation process would obtain a new Fixed and Normalised Data model in which the original `Customer` table would be divided into two tables in BCNF: `Customer (idCustomer, name, city)` and `City (city, country)`.

It is worth noting that the database administrator might prefer to keep the schema in a denormalised status for reasons of performance.

6.5 Forward Engineering Stage: Generating Restructured Database

The forward engineering stage generates the final source code of the target artefacts.

In our approach, this stage aims to either regenerate the database in a new

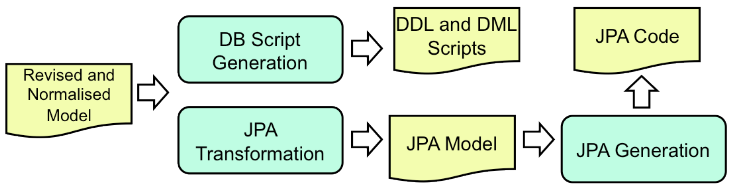


Figure 6.19: Generation of new data

database management system or to provide code for data accessing by means of specific middleware (e.g. JPA). As shown in Figure 6.19, two code generations could therefore be considered: (1) a new database schema or (2) the middleware that generates the new schema and provides access to it.

Schema generation. With regard to implementation, this is carried out using an M2T transformation that generates the DDL and DML scripts from the fixed and normalised Data model which is the output of the normalisation process. After executing these scripts a new database schema and data values are generated. This option is preferred when companies migrating data layers need to directly access their database relational schemas because they use technologies such as Oracle Forms or database stored procedures. This allows companies to have a precise control over the structures (database tables) in the new schema that have been generated.

Middleware generation. This second option allows the new database schema to be generated automatically by means of a middleware such as JPA (Java Persistence API). This allows companies to attain a higher abstraction level when manipulating databases, and the client applications can take advantage of functionalities provided by these middlewares (e.g. JPA provides object-relational mapping and automatically regenerates and synchronises the database schema). However, this comes at a price: the companies must then depend on third-party implementations. In order to implement this alternative, we have designed a two-step model transformation chain: an M2M transformation first converts the Fixed and Normalised Data model into a JPA model, and an M2T transformation then generates the middleware source code from this JPA model. Although the transformation could have been implemented in a single step by eliminating the intermediate JPA model, we have defined this step in order to make the attainment of the final M2T transformation simpler. Creating intermediate models is a well-known

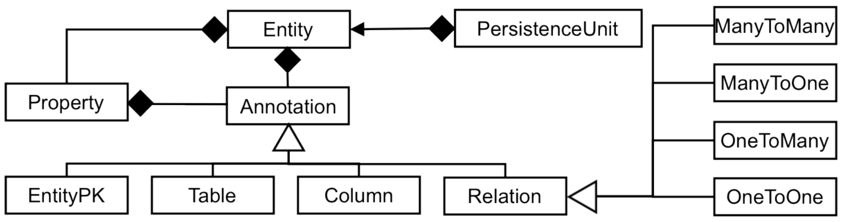


Figure 6.20: JPA metamodel simplified

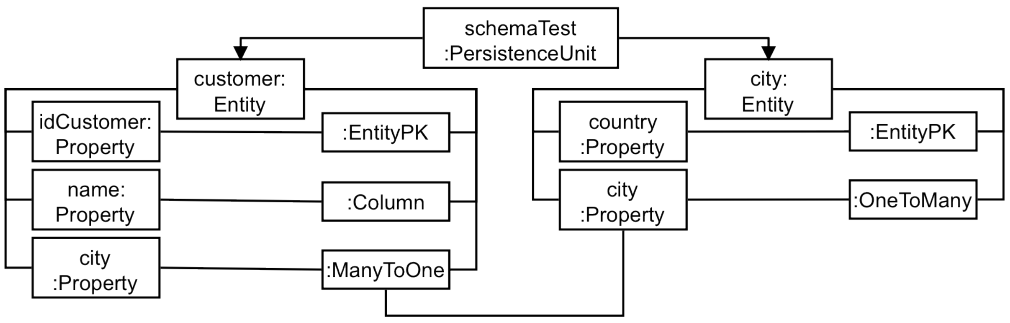


Figure 6.21: JPA model simplified of running example for `Customer` and `City` tables

technique as regards tackling the complexity of transformation by decomposing a large and complex transformation into several smaller and simpler transformations [116][117]. Figure 6.20 shows an excerpt of the JPA metamodel we have created to represent the JPA basic elements, such as entities, properties and annotations; a hierarchy is included to represent the different kinds of relations between tables (one-to-one,one-to-many, many-to-one, many-to-many).

In order to obtain the JPA model it is necessary to identify the different kinds of relations between tables by interpreting the information from the relational schema. The M2T transformation generates: (1) an entity class for each table, with attributes and the accessor methods *get()* and *set()* to provide access to each column, and (2) the annotations for the class columns which are the identifier and the relations identified.

With regard to the running example, Figure 6.21 shows the JPA model obtained during the last stage of code generation. From this model, the following artefacts would be created: (1) the JPA configuration file (i.e. persistence.xml) and (2) the following annotated classes: *Customer.java*, *City.java*, *Order.java*, *OrderLine.java* and *Product.java*. An excerpt of the code generated for the annotated class *City* is shown below in order to illustrate what code is generated by the M2T transformation:

Listing 6.8: JPA entity geenrated class *City.java*

```
@Entity
@Table(schema="TEST")
public class City implements Serializable {
  @Id
  @Column(name="CITY")
  private String city; ...
  @OneToMany(mappedBy="city")
  private List<Customer> customers; ...
  public List<Customer> getCustomers() {
    return this.customers;
  } ...
}
```

6.6 Applying our approach to the real-world case study

This section shows our MDE approach at work in a real data modernisation scenario: the Open Source Clinical Application Resource (OSCAR) [111]. The goal of this section is to demonstrate the feasibility of applying our approach. We shall show the partial and final results achieved in each stage of the data reengineering process implemented. The version of the OSCAR database schema used in this work as a case study is exactly composed of 445 tables. The results obtained after applying our approach to OSCAR are shown as follows.

Reverse engineering stage. The DAS-M strategy could not be performed owing to the size of the database (more than 8 gigabytes). Our current implementation of the DAS-M strategy is very limited because the modelling technologies (e.g. model transformation engines) are not able to deal with such large models, and several memory problems were found when using our approach. Some considerations related to these problems will be commented on Section 6.7. It was, however, possible to apply the DAS-D strategy successfully and a total of *127 foreign keys* which have a matching percentage above or equal to 90% were found, from a total of 291 pairs of columns which have a matching percentage of over 0%. In addition, 0 check constraints were found. The execution time was 3m 44s, using a server with the following specifications: *Intel(R) Core(TM) i7-3770 3.40GHz* processor and *16 GB DDR3 (1333 MHz)* of main memory. With regard to the CAS strategy, the results of the analysis performed for each step are shown below. The execution time was 9h. 42m.:

- *Step 1. Fragment model injection*: 3.533 Java files were analysed and 82.009 literal strings were found
- *Step 2. SELECT file generation*: 19.344 strings containing SQL Select code were found
- *Step 3. SELECT file generation*: 4.445 strings were considered to be SQL Select sentences, corresponding to only 221 Java files
- *Step 4. SELECT model injection*: 502 complete sentences containing 197 joins were re-created
- *Step 5. Defect model generation*: 21 foreign key candidates were found

Restructuring stage. We considered that all the undeclared foreign keys were defects and we then fixed the database schema. We encountered problems as regards identifying functional dependencies owing to the size of the OSCAR database, in which a few tables store more than 500.000 records and the average per table is almost 10.000 records. However, the ConExp tool is not able to manage such a large data input (even when modifying ConExp execution parameters such as heap memory size). We used different table sizes to analyse how the ConExp input file grows with an increasing the number of tuples in a table. We then decided to evaluate the normalisation process for tables containing 400 records, which was the largest amount of records producing ConExp input files of under 1 Gbyte in size (the greatest input file size that proves to be feasible for ConExp, after modifying the execution parameters).

We measured the execution time that ConExp requires for each individual input file. As expected, time varies depending on the file size: the bigger the size, the longer the time, but not in a lineal progression. For instance, files of about 1 Gbyte required approximately 21 minutes, whereas files of about 0,5 Gbytes required 14 minutes and those of approximately 0.25 Gbytes required no more than 10 minutes. The total time spent executing ConExp on OSCAR database was 56 hours, 43 minutes and 20 seconds.

For each of these tables we then took a random sample of 400 records and applied our normalisation process. We found 19 functional dependencies in the 445 tables of the OSCAR schema. 15 tables were affected by these functional dependencies. The decomposition algorithm produced a total of 19 new tables in the OSCAR schema (one table for each functional dependency found) and the synthesis algorithm was not needed. Each new table had its own foreign key for the table which previously contained the functional dependency.

The forward engineering stage was eventually applied and the SQL and JPA code was generated. The SQL code included the CREATE TABLE statements for the 464 tables in the new schema (445 original tables and 19 new tables) and their corresponding foreign keys. The JPA code was composed of the same quantity of annotated JPA entities (464 entities) and the corresponding configuration file (`persistence.xml`).

6.7 Assessment of the approach

Throughout this chapter we have illustrated how MDE techniques can be applied during data reengineering in a data modernisation scenario. This section analyses to what extent the use of MDE has facilitated this undertaking. To this end, Table 6.4 refines Table 6.1 by comparing for each task the traditional approach ("Traditional" column) with the MDE approach ("MDE" column). We shall first describe how each of the three stages of which a data reengineering process is composed can take advantage of MDE techniques (Sections 6.7.1, 6.7.2 and 6.7.3). We shall then calculate the gain of productivity by comparing effort required (as regards time) for the traditional and the MDE approaches (Section 6.7.4). Finally, we conclude with the benefits and drawbacks of applying MDE to a data reengineering scenario (Section 6.7.5).

6.7.1 Reverse engineering

Creating parsers

The effort involved in creating the parsers required can be significantly alleviated by using MDE tools in order to automatically generate model injectors, which combine parsing and model generation, such as textual DSL definition tools (e.g. Xtext [118]) or domain-specific languages tailored to the injection of models (e.g. Gra2MoL [47]). We have used Gra2MoL to generate parsers for DDL and DML scripts, and for Java code fragments. Injectors for Java and KDM could be reused from the Modisco framework [46]. As noted in Section 3.2, the DB-Main environment provides parsers for DDL scripts but they are built into this tool and cannot be reused.

Defining formats

Metamodelling provides a very expressive formalism with which to represent both the information to be parsed (i.e. schema and code) and the knowledge extracted. The most widely used metamodelling languages (e.g. Ecore) used to create metamodels provide greater expressiveness than the XML language commonly used to define metadata. Using metamodels in data reengineering is not a new idea. For instance, a metamodel that could be used to represent any data model (e.g. relational or object) was proposed twenty years ago [119], as was the idea of mapping between schemas (i.e. instances of the proposed metamodel). However, metamodelling was, at that time, an immature discipline and developers lacked metamodelling tools, signifying that a deductive object manager was used in [119] to define metamodels and mappings.

In our case, we have defined data and code metamodels using Ecore. Since the metamodels created represent standard languages or widely used information, they could have been available in a metamodel repository, such as Zoos [1] which is maintained by the AtlanMod group. In this case, we could have avoided the effort needed to define them. In fact, in 2003 the OMG launched the ADM initiative

[1]http://www.emn.fr/z-info/atlanmod/index.php/Zoos

[16] in order to provide a set of standard metamodels with which to represent the information commonly used in reengineering tasks, and a metamodel that could be used to represent data in software applications was included as part of the KDM metamodel. Injectors for these metamodels were also made available to the public. For instance, a KDM injector is provided by Modisco, as indicated above. However, we have not used KDM since the part of this metamodel that represents database concepts (i.e. the Data package) has some weaknesses in the context of our approach. In particular, defects, functional dependencies and DML sentences are not modelled. KDM is a very large metamodel which is extremely complicated to manage as explained in [11]. In our experience, creating one's own metamodels is more convenient than extending KDM with new elements.

Implementing data and program analysis

Representing software artefacts as models makes it possible to take advantage of model transformations to automate development tasks. Most widely used M2M transformation languages (e.g., ATL[120], QVT[39] and ETL [121] integrate declarative and imperative constructs. The objective of the former is to express mappings between the elements and queries of metamodels in models, and the latter are needed to provide support when implementing algorithmic strategies. These languages are more efficient than GPL languages as regards expressing how a target model is generated from a source model (e.g. a mapping between a UML class model and a Relational Schema model) or how a model may be refined (e.g., correcting defects in a Schema model). However, they have significant limitations when implementing more or less complex algorithms that require non trivial data structures. Since this situation is common in reverse engineering algorithms, M2M transformation languages providing a rich set of imperative constructs are needed. The traditional strategy of using a GPL to write code that analyses representations obtained by parsers and represent knowledge extracted in a format such as that of XML documents would be less effective than managing models as the input and output of the reverse engineering process since they allow us to work at a higher level of abstraction.

In our case, we have implemented the DAS-M and CAS analysis strategies by means of M2M transformations by taking advantage of the hybrid nature of the RubyTL language. Ruby code can be written at any point of a transformation because RubyTL has been implemented by embedding specially tailored constructs in order to write M2M transformations in the Ruby language. Furthermore, the DAS-D strategy has evidenced that a procedural language (i.e. PL-SQL) is more appropriate than M2M transformations as regards analysing data. This implementation has shown that employing a model injector (i.e. a Schemol transformation) may avoid the need of use a metamodelling API to generate a model as the output of the analysis.

It is worth noting that using a GPL to write model transformations for algorithmic strategies would be less productive than using embedded languages such as RubyTL since developers should manage a metamodelling API (e.g., EMF API). However, this alternative should be considered by taking in account the limitations of most transformation languages when writing complex imperative code and the immaturity of the tooling that supports these languages [5] [6].

Other Comments

Note that our solution is platform independent (e.g. tools or languages). This is because the DDL, DML and Fragment metamodels separate the reverse engineering transformation chain from a concrete platform. DDL and Fragment injectors must be created for each database technology (e.g. Oracle Forms) or programming language (e.g. PL-SQL). This implementation can be addressed using T2M languages such as Gra2Mol, rather than manually creating a parser from scratch. Both parser and model generator are produced automatically from the ANTLR grammar of the language (DDL, DML and Java fragments in our case) along with a mapping that establishes the correspondence between grammar elements and metamodel elements. In our case, the grammars are simple and the mappings are easy to write owing to the small semantic gap between the grammar and the target metamodel.

6.7.2 Data restructuring

Correcting defects in the database schema

This task has been automated by means of an M2M transformation and a wizard in order to select the defects to be corrected. Here, using M2M transformations is clearly more appropriate than creating a GPL program since a *mapping* must be implemented: the original schema is transformed into the fixed schema. It is also worth noting that our solution is according to human aware [107]. The database administrator's knowledge is considered in order to decide which of defects detected must be removed from the schema. This knowledge is represented by modifying the defect model when the administrator interacts using a wizard created for this purpose. In general, human knowledge could be expressed by either creating models or transforming previously generated models.

Detecting functional dependencies

Our approach shows the ability of models to ease the integration of third-party tools into a MDE solution (i.e. a transformation chain), specifically the ConExp tool which is used to detect functional dependencies. These integrations are achieved by means of model transformations: the output of the tool to be integrated is converted into one of the models of the chain or the input is obtained from one of the models of the chain. These two transformations are needed to integrate the tool into the intermediate point of a chain, and only one is needed if the tool is integrated at the beginning or at the end. In the case of ConExp, we have created a Gra2MoL transformation to inject FD models from a proprietary format. With regard to the input, it could be automatically generated from the DML model, but these models are very large owing to the fact that it is necessary to combine all the tuples in each table, and we have therefore created a tool (i.e. the SchemaGA driver) to automate the task of obtaining the input XML file.

Implementing Normalisation algorithms

Schema normalisation is performed by means of an M2M transformation chain that implements the decomposition and synthesis algorithms. These involve a significant amount of imperative code, including recursion (e.g. the backtracking technique is used in the decomposition algorithm). We again found the hybrid nature of the RubyTL language very useful. In a traditional approach, this task is normally implemented in the form of GPL code that manages XML documents.

6.7.3 Forward engineering

Creating the new schema

As indicated in Table 6.1, we have only addressed the regeneration of the database schema. This task has involved the creation of the JPA code that implements the data access and the SQL scripts that define the database schema. While these artefacts should be created manually in a traditional migration process, they have been generated automatically by using model transformations. For instance, JPA code has been generated automatically as explained in Section 6.5. An M2T transformation might have been sufficient to generate this code from the Data model resulting from the normalisation process. However, an intermediate M2M transformation was introduced to make it simpler to write the M2T transformation. We chose the MOFScript language to write M2T transformations. This has recently been discontinued despite being one of the most widely used M2T languages, which evidences the lack of stability in MDE tools.

The reverse engineering and data restructuring stages are independent of targets platforms. This signifies that the forward engineering stage is in charge of generating the artefacts for a particular platform, in our case JPA.

Note that the model transformation chains defined in our approach could be reused in other MDE solutions. For instance, the forward engineering chain could be integrated into a generative architecture with which to automate the creation of applications that integrate a data access layer, and the reverse engineering chain could be used into a defect analysis or metrics tool. If a model transformation chain is reused it is only necessary for its input or output model to be integrated at some point of the new solution i.e. the two metamodels involved are the same

Tasks	Traditional	MDE
T1.1	Parsers ad-hoc for DDL, DML and SQL-embedded.	Auto-generated parsers by using T2M languages.
T1.2	Using of XML Schema and XML.	Using of metamodels and models.
T1.3	Using of GPLs and XML technologies. e.g. JAXP (Java Api Xml Processing) and XQuery.	Using of hybrid languages to combine imperative and declarative bussines logic. e.g. RubyTL language.
T2.1	Implementing a wizard to assist users.	
T2.2	Integration with ConExp tool. Generating .cex files (e.g. by XSLT) and implementing and parser ad-hoc for the ConExp output (not XML).	Integration with ConExp tool. Generating .cex files from models by M2T languages (e.g. MOFScript) and using T2M languages for the ConExp output (e.g. Gra2Mol).
T2.3	Using of GPLs and XML technologies. e.g. JAXP (Java Api Xml Processing) and XQuery.	Using of hybrids languages to combine imperative and declarative bussines logic. e.g. RubyTL model-to-model language.
T3.1	Using of XSLT and XPath.	Using of M2T languages.

Table 6.4: Manual tasks vs MDE tasks

or a mapping can be established.

6.7.4 Gain in productivity

Productivity is the main factor that encourages the use of MDE [5]. Here we shall therefore present an estimation of the productivity gained by the MDE techniques. This will be done by first measuring the effort needed to implement the proposed approach, and then contrasting the result obtained with the effort involved in using the traditional approach (i.e., manual approach). In this comparison, we have identified those development tasks that are more efficiently implemented by means of MDE techniques.

Our approach has been implemented by three developers with no initial experience of MDE techniques, signifying that a 60-hour training period was initially needed. Each developer made an implementation effort of about 210 hours (6 weeks), and we have therefore calculated that a total of 690 hours would be needed to implement our solution. Tasks 1.1 and 1.2 required the definition of both the DDL and DML grammars and a grammar for a reduced subset of the SELECT SQL statement. These were not created from scratch, but were adapted from previous works taken from grammar repositories and zoos, such as the ”Software

Language Processing Suite" [2]. This adaptation and the definition of the mapping rules in Gra2Mol required a total of 120 hours. In task 1.3, the use of RubyTL eased the implementation of the algorithms used to discover the Foreign Keys. We combined model queries with the imperative functions (helper functions written in Ruby) in a declarative manner, about 110 hours. In task 2.1, the wizard implementation was performed in 100 hours. Task 2.2 was again implemented by taking advantage of Gra2Mol, even when parsing a non-XML input (which is the result of using ConExp). About 90 hours were required, along with 60 more hours to implement the decomposition and synthesis algorithms. We noticed that the hybrid nature of RubyTL again facilitated this task since both the model queries and the implementation of backtracking-based techniques such as recursion were facilitated. Finally, we used MOFScript in task 3.1 in which about 150 hours were required to implement the code generation templates.

Table 6.4 evidences that the principal reason why MDE provides a productivity gain is to the use of DSLs (Gra2MoL, Schemol and RubyTL in our case) and the fact that models provide a larger level of abstraction than formats such as XML, JSON or proprietary formats and they are very appropriate when performing tasks such as tool interoperability. The tasks in Table 6.4 in which MDE leads to more productivity than a traditional approach are shown as follows, along with a brief justification of why this gain is obtained.

- We used Gra2MoL, which automatically generates injectors from declarative rules that establish a mapping between grammar elements and metamodel elements, in tasks T1.1 and T2.2. A detailed explanation of the benefits of this language versus the alternative of manually creating parsers is shown in [47]. Some data engineering tools (e.g. DB-Main) have the built-in parsers required in data migration but they are part of a proprietary solution and they cannot be used separately.

  In [122] it is calculated that the effort needed to learn Gra2MoL and Ruby is similar (although, according to the effort estimated in [123], Gra2MoL is 1,64 times more difficult to learn than Java). However, the effort required

[2]http://slps.github.io/zoo

to implement parsers and model generators by means of Gra2Mol is considerably less than that involved in writing them by means of a GPL such as Ruby or Java owing to the fact that the size of a Gra2MoL program is quite a lot smaller that the size of injectors written with a GPL. For example, in order to create a DDL injector in Gra2MoL it was necessary to program 783 LOC (corresponding to 131 lines for the DDL metamodel, 150 lines for DDL grammar and 502 lines for mapping rules). On the other hand, the parser and model generator automatically generated by Gra2Mol are composed of 10,448 LOC in Java language, which means 92,50% more effort than when using an MDE technology such as Gra2Mol.

- Writing the algorithms involved in reverse engineering and normalisation (tasks T1.3 and T2.3) is more effective when using models and a model-to-model transformation language (i.e. a DSL) than when using an XML format and GPL code. This second option, which is commonly used, implies programming at a lower level of abstraction and uses XML technologies such as JAXP and XPath. As noted in [122], RubyTL is easier to learn than Gra2MoL and moreover, there is no need to use APIs to manage models (in the case of XML documents is needed to use JAXP).

- Models facilitate tool interoperability as discussed above for task T2.2. MDE techniques allow us to bridge the interoperability gap at the syntactic and semantic levels, as explained in [34].

We have estimated the following cost for the traditional and MDE implementation. The cost for the traditional solution has been estimated using the code lines generated in each case.

- 240 hours for the three injectors required in tasks 1.1 and 1.2. When using MDE, we performed these tasks in 120 hours.

- 140 hours for the algorithms that identify the defects, which is a little more than when using MDE (110 hours).

- 160 hours for the integration of ConExp and 80 hours for the normalisation algorithms. As stated above, when using MDE we required a total of 150 hours.

We therefore estimate that a traditional strategy for our approach would require 240 hours more than using an MDE solution, which supposes 34,78% more effort.

6.7.5 Benefits and drawbacks

According to the above discussion and our previous experience in, for example, the works presented in [11] [47] [9], we have identified the following main benefits of using MDE in data reengineering, some of which are also applicable to software reengineering.

- *Productivity.* A productivity gain is achieved principally because using DSLs reduces the effort required to perform two tasks: creating parsers and generating software artefacts, which should be manually implemented. In addition, model transformations are more appropriate than GPL code when expressing the mappings involved in data reengineering. There are many domain-specific languages that are specially tailored in order to write model transformations (see Section 2.3).

- Expresiveness. Metamodelling is a more expressive formalism than XML and JSON when representing the information involved in a data reengineering process. Rather than using proprietary formats, models allow the information to be uniformly represented, which favours software quality, e.g. interoperability, extensibility or reuse. The existence of widely adopted metamodelling languages (e.g. Ecore) strengthens the benefit of metamodels with regard to proprietary formats.

- Easy tool integration. Models ease the integration of tools/solutions that are available to support some of the tasks to be performed. The input/output artefacts of software tools are injected/transformed into models in the MDE solution.

- Database and software evolvability. As stated in [4], the integration of software and database evolution processes is one of the current challenges to be addressed by the reengineering community. They usually evolve independently thus leading to inconsistencies resulting in high costs of software and data maintenance. Using MDE build software could improve the capabilities of the integration between the software and database subsystems, i.e. a model transformation chain could generate the changes in the target artefact from a model representing the changes in the source target.

- Consistency. Another challenge identified in [4] is a lack of synchronisation among the database schema changes and code that accesses data by means of an Object-Relational Mapper (ORM). In software evolution, development teams sometimes work in an undisciplined manner (i.e., database evolves without considering the ORM definitions and vice versa). Again, a transformation chain could allow the consistency between database and data access code to be maintained.

- Reuse of solutions. Metamodels and model transformation chains can easy be reused in another different solution to that for which they were created. In order to reuse a chain, the inputs and outputs of this transformation chain must be connected to another solution by injecting/transforming them in accordance with the proper formats required by the other solution. This is a special case of the integration benefit annotated above.

- Platform independence. Independence of source and target technologies can be achieved the use of models. This would be a particular case of reuse: a solution is adapted to specific platforms by means of M2M transformations. An M2M transformation converts a source platform model into the solution's input model and (ii) source platform artefacts are generated in two steps: an M2M transformation firstly converts the solution's output model into a target platform model, and an M2T then generates target software artefacts.

- Standarisation. A set of standard metamodels could be defined to represent the information commonly used in data reengineering tasks (e.g. a DDL Schema), in order for them to be reused in different applications. ADM is an initiative with this purpose, as explained at the beginning of this section. Moreover, injectors for these metamodels could be made publicly available.

We have also encountered some drawbacks which could make the adoption of MDE techniques in reengineering difficult.

- Scalability. It is often necessary to manage large models (e.g. database models) in a reengineering process. Although several model repositories are emerging which are intended to efficiently manage large models [115] [114], this technology is not as yet sufficiently mature.

- Tool maturity. While MDE has been the focus of a great deal of academic interest over the last decade, its adoption by industry is far from having been achieved [5]. Except for some DSL definition tools, most MDE tools and environments lack the robustness and level of maturity required to allow the developer's productivity to be similar to that achieved with traditional tools. This limitation is particularly evident in model transformation engines which are usually discontinued and most of which appear as prototypes of research tasks (e.g. MediniQVT [3] or RubyTL).

- Lack of standards. The KDM standard metamodel provided by ADM to represent, the code and data involved in software systems at different levels of abstraction is very large, and this makes it difficult to understand [47]. As explained at the beginning of this section, the Data package requires more expressivity to model all aspects addressed in our approach.

- Lack of metamodel and injector repositories. Although metamodels for languages which are frequently involved in reengineering processes (e.g. PL/SQL and COBOL) are publicly available, there are no injectors for them and they must be created in each project. Moreover, the existence of different versions of the languages and platforms makes it difficult to have a repository that contains the metamodels (and the corresponding injectors) for them.

- Transformation expresiveness. It is often necessary to implement more or less complex algorithms, which requires the definition of non trivial data structures (e.g. using graphs or recursion for the backtracking technique).

[3]http://projects.ikv.de/qvt/

Most M2M transformation languages are not appropriate for the implementation of these algorithms, which usually have to be written in a GPL code (e.g. Java) that uses a model management API (e.g. EMF). It is not therefore possible to take advantage of the declarative constructs of most M2M transformation languages, e.g. declarative model navigation languages.

Finally, it is worth noting that our work has not addressed data and program conversions. With regard to data conversion, there are a number of commercial ETL tools that support the automation of this task, which are used by companies involved in data migrations. Program transformation languages have, meanwhile, been specially designed to specify program conversion and may be useful in data reengineering, as noted in [17]. However, model transformation languages are more appropriate when a complex reverse engineering must be performed (e.g. the layout inference described in [9]). Moreover the integration and scalability of program transformation systems are still challenges to be addressed [124]. With regard to the existing proprietary tools for software (e.g. DMS [4]) and data reengineering (e.g. DB-Main), the problem is the variability management and the difficulties involved in adapting these tools to concrete problems (e.g. migrations).

6.8 Conclusions

We have presented an MDE-based reengineering approach with which to tackle the well-know problem of recovering implicit information from relational data in order to elicit defects in the database schema design. In particular, we have illustrated our approach with two kinds of defects: undeclared or disabled foreign keys and disabled check constraints. In order to create the new fixed schema, we apply a normalisation if necessary. Finally, the new schema is created by generating either DDL scripts or JPA code. The approach has been validated in a real data modernisation scenario and we have rigorously assessed the approach by comparing how the tasks performed in a data reengineering process can be automated by means of MDE techniques and tools. We have concluded by summarising the benefits and drawbacks that we have identified. When MDE is applied to data

[4]http://semdesigns.com/Products/DMS

reengineering, some specificities are evidenced regarding model-driven software reengineering, but the majority of concerns are common to both areas.

Next, we will present an important lack we found as resulted of applying our reengineering approach and which motivates the G3 goal of this thesis and the works presented in next Chapter. Finally, we will outline the challenges to be tackled by MDE in the data migration context.

After more than a decade since the emergence of MDE (e.g. MDA in 2000, ADM in 2003), the industrial adoption of this new technology is still very limited. In [125], the results of a survey of MDE practitioners were reported and they evidenced that MDE is principally being adopted to create small domain-specific languages, which automate some development tasks, rather than being used to build whole systems or reengineering legacy systems. However, poor tool support is still one of the main limitations as regards achieving the adoption of MDE, as noted in [5] [126]. The success of a new development paradigm requires robust, usable and efficient tools and environments that are able to support software development processes. As noted in [126], MDE tools need to be more resilient and simple since ”MDE can work but it is a struggle”. As evidenced in our work, MDE techniques can be very useful in data reengineering to represent harvested knowledge in reverse engineering, automate the creation of parsers or generate target system's artefacts, among other benefits. But the lack of appropriate tools has led to a reduced impact of MDE in the software and data reengineering areas, which require sophisticated tools. It is necessary for developers to have environments that provide MDE tooling (e.g. injectors for different source artefacts and appropriate model transformation languages). In addition, these tools should be easily integrated with existing reengineering tools.

The main challenges in data-intensive evolution are discussed in [4]. MDE techniques could be useful for tackling some of them. For instance, tools developed for the automation of software and data reengineering tasks could be integrated by means of models, as explained in this thesis as regards integrating ConExp into a model transformation chain. Models could also be useful to support traceability between the object-oriented schemas managed by object-relational tools (ORM) and a database schema in order to avoid inconsistencies.

"I slept and dreamt that life was joy. I awoke and saw that life was service. I acted and behold, service was joy."

Rabindranath Tagore
(Suggested by Ricardo and Laura)

7 Migration Tool

This chapter is devoted to present the tool built for supporting the definition and enactment of our data reengineering process. As motivated in Chapter 1, the tool is intended to support model-driven reengineering process (e.g. migrations), in general. We shall firstly introduce the scenario of the tool, identifying its inputs and outputs. Next section will present the running example which is the data reengineering process presented in Chapter 6. Then, three sections will deeply describe the main requirements of a migration tool: process definition, instantiation and enactment. Following, we shall explain how to use the tool in a software migration and then, we will show the results of applying the tool to our case study. This will provide an assessment along with a set of lessons learned that will be summarised in the corresponding section. We will end the chapter presenting the conclusions of the tool built.

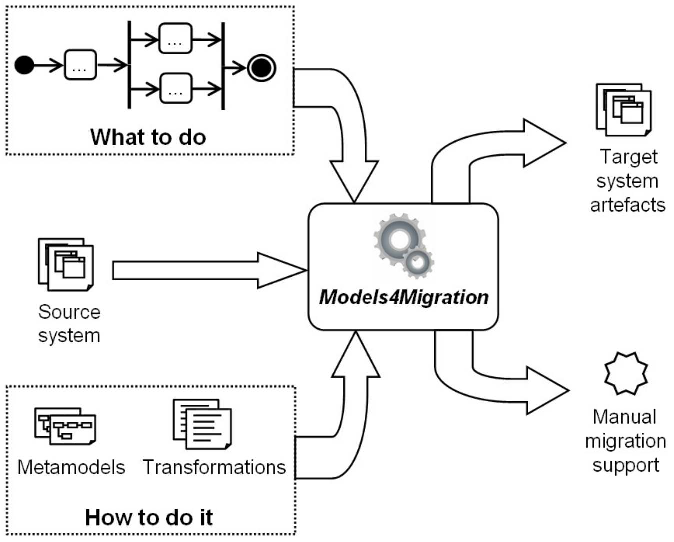


Figure 7.1: Scenario of the tool.

7.1 Introduction

The tool presented in this chapter is aimed to perform MDE-based migrations. So, we do not address software migrations in general, but those relying on MDE solutions. The scenario of use of the tool can be therefore seen in Figure 7.1.

The starting point of the scenario is a source system (the entire system or a part of it) that needs to be migrated, and a set of MDE resources (i.e. metamodels and transformations) aimed at performing the migration from one technology to another one. For example, we could think of a Oracle Forms application that has to be migrated to Java, and we have developed a set of transformations to take Forms source code files and generates Java code files.

In order to migrate the source system, some transformations have to be applied on certain resources, and some transformations have to be executed after others. This is, the execution order of the transformations has to be defined, as well as the artefacts, tools and resources that are the required input to execute the transformations. It can be seen that there is a workflow of transformations that defines what it must be done to migrate some artefacts, whereas the metamodels and transformations indicate how to perform the steps of such migration.

It is desirable to specify this workflow with independence of the concrete source application so it can be reused in different source applications, and it would be interesting to have some graphical facility to define such workflows.

Then, the inputs of our tool are:

- The artefacts of the source system.
- The metamodels and transformations to migrate the source system (they must be available).
- The workflow that indicates how to apply the transformations to the source artefacts (actually it will be created with a DSL provided by Models4Migration).

And the output of the tool are:

- The artefacts of the target system that can be automatically generated.
- The tickets for manual tasks which are integrated into a task management tool.
- Some facilities that are integrated in a development IDE (e.g. Eclipse) to assist developers in the manual migration of some parts of the system.

7.2 Running Example

As said in Section 1.3, our research group collaborated with a software company in a pilot project aimed at applying MDE techniques in the migration of Oracle Forms applications to the platform Java (Java Swing and JPA). The Models4Migration tool was developed to support the model-driven reengineering process defined in such a migration project. This process involves manual, automated and semi-automated tasks and the automation is achieved through several chains of model transformations that generate some artefacts of the target system (e.g. code of presentation and persistence layers). With regard to the data layer, these meta-models and model transformations correspond to the data reengineering process presented in Chapter 6.

Hence, the database schema conversion proposed in this thesis is an appropriate context to define a running example which can illustrate the explanations of the

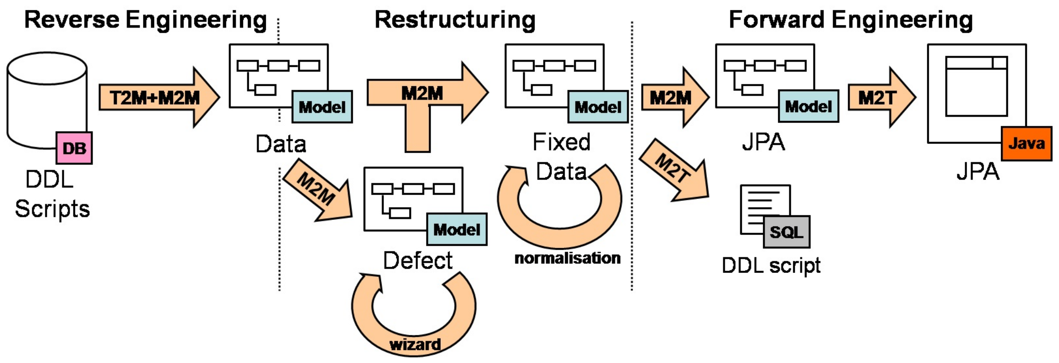


Figure 7.2: Migration plan of the running example.

tool that we shall include in this chapter. The running example presented in Section 6.2 will be also taken, where JPA (Java Persistence API) is used as the target framework for the persistence layer. Figure 7.2 shows a resume of the model transformation chain defined in Chapter 6 to transform the legacy database schemas (i.e. DDL/SQL scripts) into the JPA artefacts and the DDL script which regenerated the new database schema.

Throughout this chapter, a simplification of the database schema example used in Section 6.2 will be used here (we shall omit the `OrderLine` and `Product` tables due to more complexity do not contribute to clarify this chapter). However, in this We introduce

Listing 7.1: Database schema of the running example for the tool

```
TABLE Customer (idCustomer, name, city, country)
  PK<idCustomer>
TABLE Order (idOrder, date, idCustomer)
  PK<idOrder>
```

This schema presents a potential defect that could be fixed. The defect is that the `Order` table contains an attribute that has the same name that the primary key (PK) of the `Customer` table (`idcustomer`), nevertheless there is no foreign key (FK) from `Order` to `Customer`. Sections 7.3 and 7.4 show how our approach deals with this running example.

7.3 Process Definition: A DSL for migration processes

The emergence of the SPEM language has caused an increased interest in applying MDE in process engineering, and most of the approaches to model software processes are based on this OMG standard. However, these works have not had an impact on the industry yet. As explained in Chapter 3, SPEM 2.0 does not address the enactment of process models, but SPEM models must be linked to behaviour models expressed in other formalisms (e.g. UML activity diagrams). To overcome this lack of enactment, some SPEM extensions have been proposed in order to provide a Process Modelling Language with capabilities of executing process. As discussed in Section 7.3, currently available SPEM extensions are not suitable for the kind of enactment required in our approach, which must automatically execute model transformations and manage manual tasks by means of tools which are widely used by software companies, such as Trac and Mylyn. We therefore decided to create a new DSL whose metamodel is based on SPEM. Particularly we have adopted the SPEM core concepts (tasks, roles, work products, tools, etc.) and we have extended them to fit the software migration domain. For enacting the models we have implemented used activity diagrams.

In this section we will present the *Migration* metamodel proposed to define migration processes, which has been created as an extension of SPEM. We will pay special attention to describe the elements defined to provide us the level of detail needed to enact migration process models. A graphical concrete syntax has been defined for this metamodel with the aim of supporting designers when modelling a migration plan (e.g. the Abstract Migration model shown in Figure 4.2). Creating these models by hand can be tedious and unproductive and the *Migration* DSL greatly facilitates this task. The DSL concrete syntax is based on the SPEM notation which is increasingly used by the software community, and so the designers which are used to SPEM to quickly master our DSL. With regard to the DSL semantics, this is given by means of a interpretation process. We will introduce the DSL notation in this section through a process model of the running example, while that the process interpreter is described in Section 7.5.

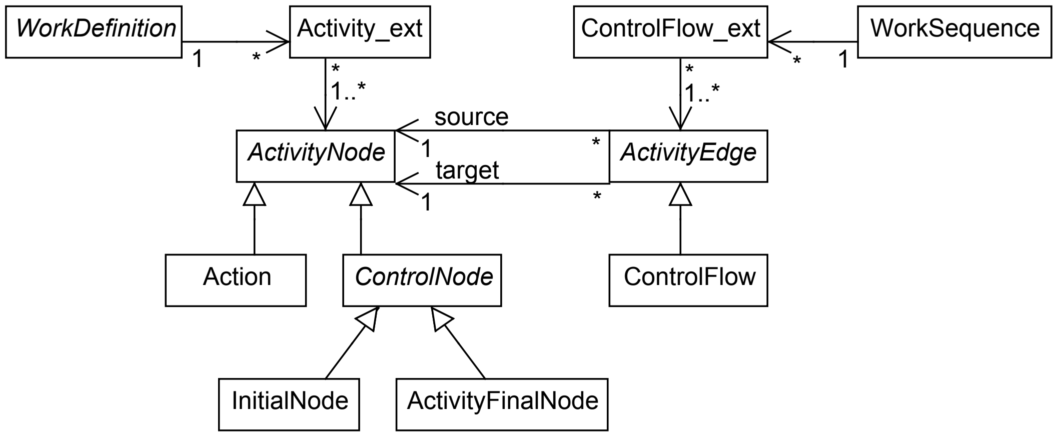


Figure 7.3: Fragment of the UML2 Activity Diagram

7.3.1 The Migration Metamodel

A concrete model instantiates an abstract model by providing the information needed for enacting the process. Both models can be defined in terms of the same concepts, so a unique metamodel can be used to represent them. As illustrated in Figure 4.2, this separation between abstract and concrete model is also needed in migration processes, as a particular case of software processes.

The standard metamodels (e.g. UML and KDM) are usually too large and complex and theirs extension mechanisms are not enough to the needs of designers. For instance, the use of KDM standard [77] was discarded for implementing the reverse engineering stage in [9]. In the case of the *Migration* metamodel, we have also found that SPEM is not appropriate due to the reasons explained in the previous section, and a custom metamodel would better meet our requirements. However, instead of starting from scratch, we have created a metamodel based on SPEM with the aim of reducing the effort required in its definition, as well as to take advantage of the benefits and experience captured in this OMG standard specification. We have reused the SPEM metamodel, and we have defined new elements in order to represent specific concepts of the MDE migration processes. More specifically, we have reused the elements in the excerpt of SPEM shown in Figure 2.5.

As shown in Figure 7.3, we have used the `Activity_ext` and `ControlFlow_ext` proxy elements to connect UML2 Activity Diagrams to our metamodel, through of the borrowed SPEM elements, in order to provide it with behaviour modelling concepts. The Activity_ext and ControlFlow_ext elements are proxies that refer to the ActivityNode and ActivityEdge of the Activity Diagram.

Figure 7.4 shows the Migration metamodel that we have devised. This metamodel has been designed to deal with migrations that can be both partly automated by means of MDE techniques and manually implemented at some parts. To achieve the level of detail required for the enactment of our approach, we have defined classes which represent the concepts of a MDE migration, such as *TransformationChain*, *M2MDefinition* or *M2MEngine*, and they extend SPEM classes. We have only extended the ProcessStructure package since it allows the definition of ordered tasks; the MethodContent is omitted, given that it is not useful for our solution because only a fixed set of elements is needed as our approach is intended to define only software migration processes. Basically, the extension of SPEM consists in adding three main concepts by defining three classes: MigrationTaskUse, MigrationToolUse and MigrationWorkProductUse (see Figure 7.4a). These classes extend the classes defined in SPEM's ProcessWithMethods package, which refines metaclasses in the ProcessStructure package in order to relate it to the MethodContent package. Each one of the new classes is explained below.

MigrationTaskUse (see Figure 7.4b) is a TaskUse that is the root of the hierarchy of migration tasks. Each task can be Automated, Manual or Semi (semi-automated).

- A Transformation is a task which takes some input and generates some output. We have defined a subclass for each kind of transformation:
  - Injection is a transformation that obtains models from source files, so the models mirror the structure of the source language,
  - M2M is a model-to-model transformation and
  - M2T is a model-to-text transformation (code generation).
- A TaskChain is a task that is composed of one or more sub-tasks. A TransformationChain is a subtype of TaskChain that is composed of Transformations.

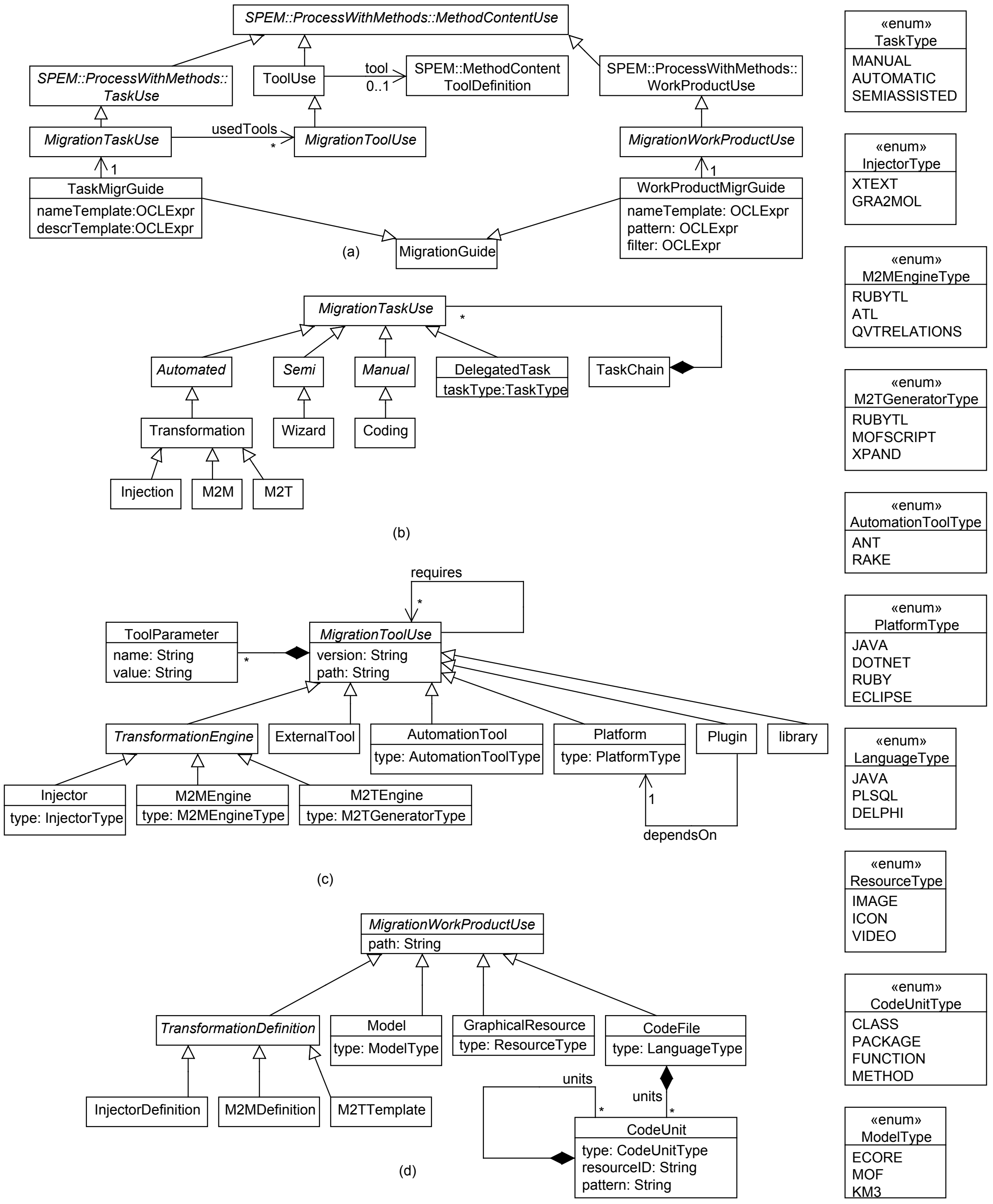


Figure 7.4: Migration metamodel

- A `DelegatedTask` is a task that is accomplished by an external tool or a third-party application. This type of task needs an attribute to explicitly indicate whether the task is automated, manual or semi-automated.

- Wizard represents a task that prompts an user some input in order to accomplish some other tasks (in order to create semi-automated tasks).
- A Coding is a kind of task that represents manual coding tasks, such as manual translations of a source artefact to a new one in the target technology or, basically, manual programming in order to create new artefacts.

MigrationToolUse (see Figure 7.4c) inherits from ToolUse, which is not included in the SPEM specification, but extends the MethodContentUse element for the sake of consistency. The separation of the ToolDefinition and ToolUse concepts stems from the fact that a concrete tool (e.g. a Java Virtual Machine | JVM) must be separated from a concrete use of the tool (e.g. the standard JVM included in the JDK 1.7 that is accesible from certain local path). A MigrationToolUse can have a version, a path and some parameters (ToolParameter). Some concrete subclasses of MigrationToolUse are the following:

- Injector, M2MEngine and M2TEngine are tools that can perform the Injection, M2M and M2T tasks, respectively.
- AutomationTool is a tool that can generate a new software artefact based on some inputs. Such a tool could also prompt developers to provide some kind of data.
- Platform, Plugin and Library are used to represent the software dependencies that are required for a tools to be executed properly.

MigrationWorkProductUse (see Figure 7.4d) is a specialisation of WorkProductUse. Some concrete subclasses of MigrationWorkProductUse are the following:

- InjectorDefinition, M2MDefinition and M2TTemplate represent the transformation definitions (e.g. transformation files), and are associated with the Injection, M2M and M2T tasks, respectively.
- Model represents a model artefact.

- `CodeFile` can be used to represent any source file, and significant `CodeUnits` can be defined for it. For example, a Java file (`CodeFile`) can contain one or more classes which can contain several methods (`CodeUnit`). The `type` of the units and the `pattern` that matches them with EBNF notation can be defined if needed.

- Other work products can be modeled with dedicated classes, such as `GraphicalResources` for modelling images.

The Migration metamodel defines *use* relationships between each kind of `Transformation` and its corresponding `TransformationEngine` (a type of `MigrationToolUse`) and `TransformationDefinition` (a type of `MigrationWorkProductUse`). Concretely, `Injection` is related to `Injector` and `InjectorDefinition`, `M2M` is related to `M2MEngine` and `M2MDefinition`, and `M2T` is related to `M2TEngine` and `M2TTemplate`; these relationships have been omitted in Figure 7.4 for the sake of clarity.

Finally, a `MigrationGuide` indicates how to instantiate an element in the Concrete Migration model from one or more elements from the source application (greater detail on this is provided in Section 7.4.3). Two subclasses of `MigrationGuide` have been defined:

- `TaskMigrGuide` defines the template to create the concrete task names (`nameTemplate` attribute) and descriptions (`descTemplate` attribute).

- `WorkProductMigrGuide` is more complex, and it allows the definition of `patterns` and `filters` to identify the artefacts (or parts of the artefacts) to be migrated from the source application, in addition to the name of the concrete work product names (`nameTemplate` attribute).

As mentioned before, the graphical notation of MigrationDSL is rather similar to the one of SPEM. Table 7.1 shows the notation for each element included in MigrationDSL (see Figure 7.4). Some elements of the UML Activity Diagrams, such as the initial state, have been omitted in the table. More information about bridging abstract (metamodel) and concrete syntaxes (notation) can be found in [127].

Concrete syntax	**Abstract syntax**
	`Activity`.
M2M	`MigrationTaskUse`. For each type of task, a different label is included in the icon. These labels are: *Chain* (`TaskChain`), *Inject* (`Injection`), *M2M*, *M2T*, *Deleg* (`DelegatedTask`), *Wizard* and *Coding*. `DependencyContraints` have no graphical notation but they are defined as properties of `Coding` tasks.
m2m	`MigrationToolUse`. For each type of tool, a different label is included in the icon. These labels are: *inject* (`Injection`), *m2m* (`M2MEngine`), *m2t* (`M2TEngine`) *extern* (`ExternalTool`), *autom* (`AutomationTool`), *platform*, *plugin* and *library*.
M	`MigrationWorkProductUse`. For each type of work product, a different label is included in the icon. These labels are: *inject* (`Injection`), *M2M* (`M2MDefinition`), *M2T* (`M2TTemplate`), *M* (`Model`), *res* (`GrahicalResource`), *C* (`CodeFile`) and *U* (`CodeUnit`).
	`RoleUse`.
	`MigrationGuide`.
◆—	Containment between `BreakdownElements` (such as `TaskUses`) and `Activities` (i.e. used to represent the `TaskUses` contained in `Activities`). Also used to represent the containment of `CodeUnits` inside `CodeFiles`.
↳ in	`ProcessParameter` of type `in` that represents a `WorkProductUse` as an input of a `TaskUse`.
↳ out	`ProcessParameter` of type `out` that represents a `WorkProductUse` as an output of a `TaskUse`.
→	A control flow between two `Activities` or between two `TaskUses`.
↳	A dependency between `TaskUses` and `ToolUses`, or between `Taskuses` and `RoleUses`.

Table 7.1: Graphical notation of MigrationDSL for defining MDE migration processes.

7.3.2 Application of MigrationDSL to the running example

We have modelled the data migration process defined in the running example (see Figure 7.2) with MigrationDSL and the resulting Abstract Migration model can be seen in Figure 7.5. The `MigrationGuides` and some input and output `WorkProductUses` such as the metamodels and the model transformation definitions, have been omitted for the sake of clarity. The running example is composed of three activities (see Figure 7.5a):

1. *Data Inject* injects a model from a legacy database schema; the resulting model can be analised by means of model transformations (see Figure 7.5b).
2. *Data Restructuring* checks and fixes the model obtained in the previous step (see Figure 7.5c), including a database normalisation task if needed (see Figure 7.5d).
3. *Code Generation* generates JPA code to create and access to the migrated database DDL Scripts that regenerates diectly the new schema in a database (see Figure 7.5e).

These activities involve both automated (*Inject DDL*, *Abstract Data* and other *M2M* and *M2T* transformations) and semi-automated (*Confirm Errors Wizard*) tasks. Each task references the tools it uses: `Injector`, `M2MEngine` or `M2TEngine`, which respectively have *GRA2MOL*, *RUBYTL* and *MOFSCRIPT* as their `type`. The generated work products are:

- Models: *DDL Model*, *Data Model*, *Defect Model*, *Data Fixed Model*, *FD Model* and *JPA Model*.
- Input artefacts obtained from the legacy system: *DDL Script*.
- Output artefacts of the migrated system: *JPA Code* and `CodeFiles` which contain `CodeUnits` such as `getColumnMethod` to refer to each JPA method, and DDL script which contains SQL sentences to regenerate the new schema in a database.

It would be necessary to explain briefly the database normalisation step shown in Figure 7.5d. When a fixed schema is obtained (on the *Data Fixed model*), it is possible to configure our migration process to normalise the new schema. In this case, the normalisation level is checked on the schema and, if needed, the

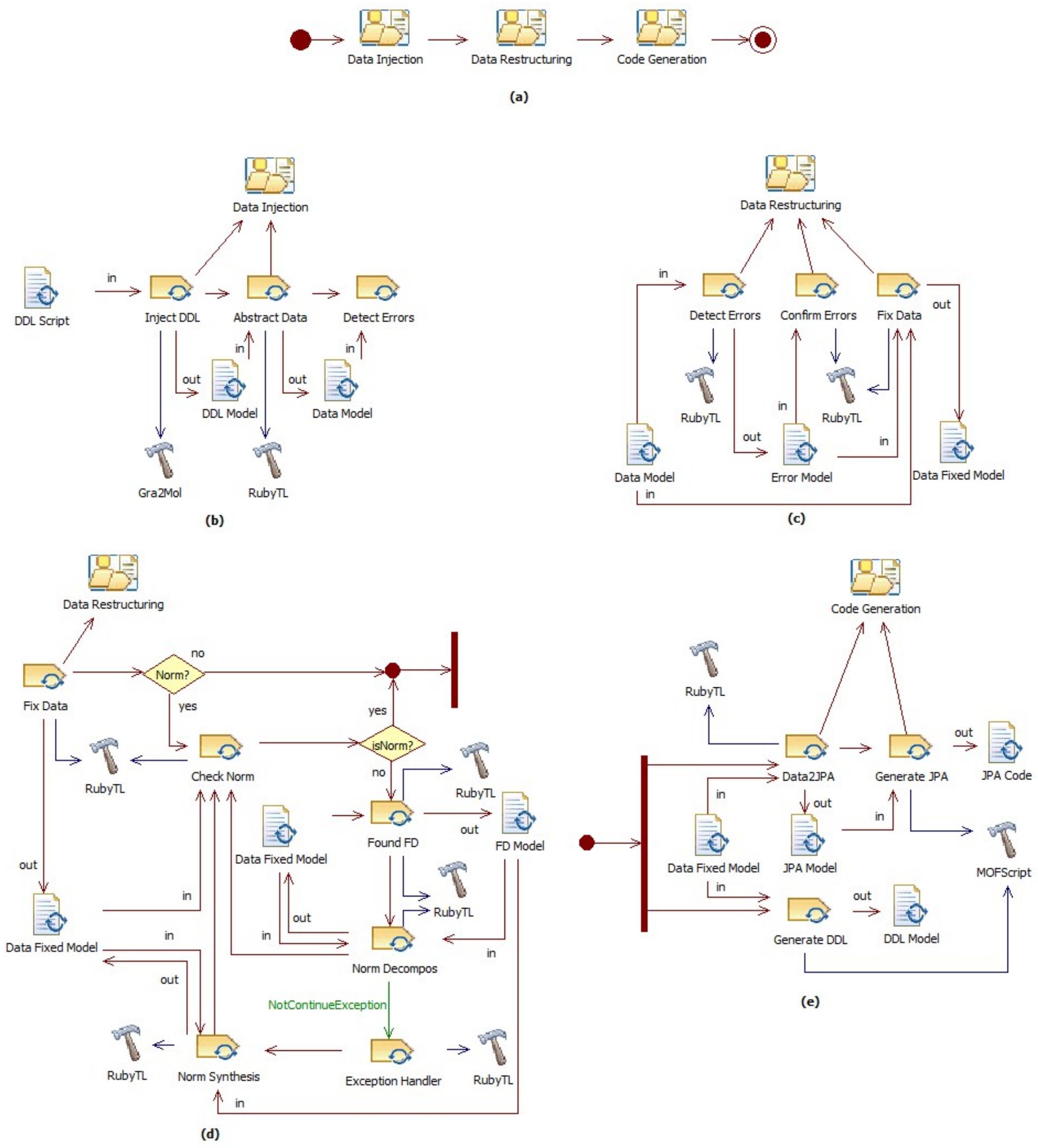


Figure 7.5: Abstract Migration model example

functional dependencies are found (*FD model*) and normalisation algorithms are applied (normalisation by decomposition algorithm -*Norm Decompos* task- or by synthesis algorithm -*Norm Synthesis* task- if the first fails).

7.4 Process Instantiation

This section describes how a Concrete Migration model is instantiated from an Abstract Migration model. Firstly, the instantiation process is illustrated; then, the Inventory metamodel is described; the use of the `MigrationGuide` class is explained in detail next and finally, the Concrete Migration model for the running example is described.

7.4.1 Instantiation of Concrete Migration models

The instantiation process has been implemented by means of an M2M transformation, as indicated in Section 4.4 (see Figure 4.2), which has the Abstract Migration and an Inventory model as inputs.

Since an Abstract Migration model is a general description intended to migrate any application that conforms to a source technology, the work products involved in the migration are expressed as variables instead of actual artefacts. For instance, the *DDL Script* work product of the running example (see Figure 7.5) is used as a variable that denotes any script that defines a legacy schema. Therefore, the instantiation of a Concrete Migration model from an Abstract Migration model requires replacing these variables with actual resources of the application to be migrated. Moreover, an Abstract Migration model may define fixed work products that must be included in the Migration Cartridge (as said in Section 4.4.4). The instantiation of a Concrete Migration model also involves the generation of one or more tasks for each concrete work product that has replaced the variable that represents the input of that task. For instance, in the running example, an *Inject DDL* task is created for each concrete file that replaces *DDL Script*.

The resulting Concrete Migration model is used by our tool to generate the actual tasks that must be executed or manually completed by developers. As commented in Section 4.4, the M2M transformation that performs the instantiation also requires an Inventory model that provides information about the resources of the application to be migrated.

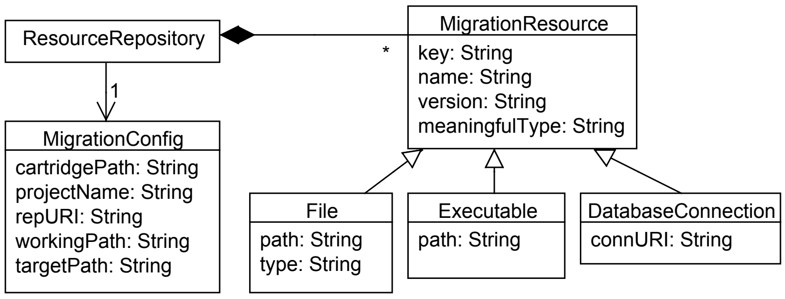


Figure 7.6: Inventory Metamodel

7.4.2 Inventory metamodel

According to the metamodel shown in Figure 7.6, an Inventory model defines a collection of `MigrationResources`. `MigrationResource` represents the artefacts or work products of the application, and its instances have a `key` to identify it, a `name` of the file, the `version` of the resource, and a `meaningfulType` tag that explicitly indicates the purpose of the file (it is useful when files of the same type can be used for different purposes). A hierarchy of subclasses of `MigrationResource` represents the different kinds of artefacts such as `File`, `Executable` and `DatabaseConnection`. A `File` is specified with the `path` and `type` of the file, an `Executable` just need the `path`, and a `DatabaseConnection` must include all the information required to connect to a database (`connURI` attribute).

Every Inventory model must also specify a set of parameters (`MigrationConfig` class) needed for the enactment such as the path to the Migration Cartridge (`cartridgePath` attribute), the name of the generated Eclipse project (`projectName` attribute), a connection string (`repURL` attribute) to the Trac repository and some other paths as working and target directories. In the running example, the Inventory model would specify all these parameters plus a database connection (instance of `DatabaseConnection`) to the new system and the path to the DDL file (instance of `File`) containing the definition of the legacy relational schema. Noting that these parameters depend on the kind of enactment supported.

The Models4Migration tool displays wizard to make the creation of an Inventory model easier, allowing developers to select the resources needed for the migration and automatically generating the Inventory model. This wizard is generated at runtime: any work product of the Abstract Migration model that is not fixed or generated by a task is automatically recognised and requested by the wizard; paths to the tool engines used in the process are also requested.

7.4.3 MigrationGuides

An Inventory model that includes all the elements needed to instantiate a Concrete Migration model from an Abstract Migration model may be too large and its creation would be a tedious and error-prone task for the developer. Therefore, the Migration metamodel (see Figure 7.4) defines `MigrationGuides` to support the automatic inference of resources and also the generation of concrete work products and tasks in the Concrete Migration model.

Each `WorkProductUse` or `TaskUse` in an Abstract Migration model can be annotated with a `WorkProductMigrGuide` or `TaskMigrGuide`, respectively, establishing how a concrete work product or task is generated from an abstract one, which may include inferring information from the source application. `MigrationGuides` guide the instantiation transformation by indicating:

1. which resources must be inspected in order to generate a concrete work product or task from an abstract one,
2. how these resources are obtained and
3. how the attributes of the generated elements are filled.

The instantation transformation uses `MigrationGuides` to inspect the artefacts specified in the Inventory model in order to generate the Concrete Migration model by performing the actions indicated above on the abstract model: replacing the variables denoting work products by references to concrete artefacts and generating a concrete task for each input artefact.

One concrete task is instantiated for each combination of an abstract task defined in the Abstract Migration model and a concrete work product that has been generated from an input of that task. Each instantiated concrete task has a different name and its inputs and outputs refer to the instantiated `WorkProductUses`

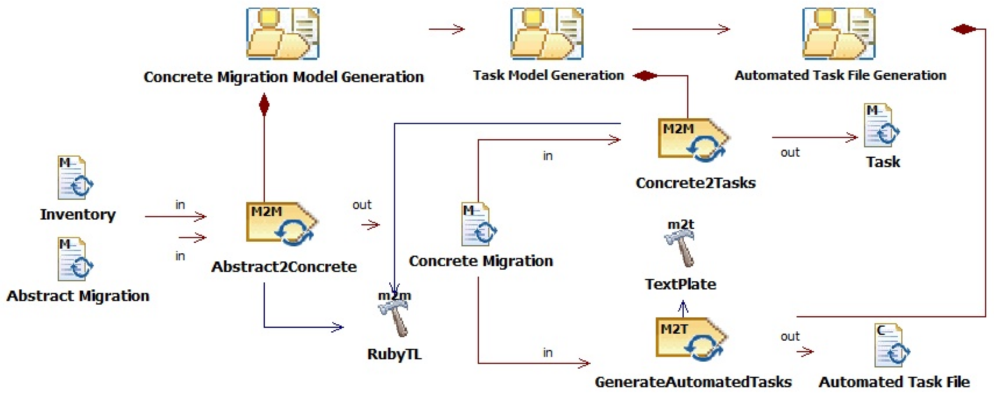


Figure 7.7: Process Interpretation model.

(e.g. concrete work products) instead of the abstract (variable) ones. `TaskMigrGuide` provides an expression to infer the name of each generated task.

7.4.4 Concrete Migration model for the running example

A Concrete Migration model can be instantiated from the Abstract Migration model of the running example (see Figure 7.5). With regard to work products, the instantiation distinguishes between existing work products which are an input to a task (e.g. *DDL script*) and work products generated as result of executing a task (e.g. *DDL model*, *Error model* and *JPA code*). For the former, the information to fill the generated work products (e.g. paths and file names) is obtained from the Inventory model; for the latter, the information is obtained by means of a `WorkProductMigrGuide`. For instance, *JPA Code* would be transformed into many concrete Java files that correspond to every JPA class created during the process. One of these Java files would be *Order.java* which defines the *Order* class with JPA annotations and accessor methods such as the *getIdOrder()* method, that is derived from the *JPA Code* and *getColumnMethod* variables, respectively.

Regarding to the generation of concrete tasks, each abstract task in the Migration model is instantiated in a set of concrete tasks, one for each input existing artefact. Each instantiated concrete task has a different name and its inputs and outputs refer to the instantiated `WorkProductUses` (e.g. concrete elements) instead of the variables. The name of the tasks is obtained by means of the information provided by a `TaskMigrGuide`. In the running example, an *Inject DDL* task is generated for each existing *DDL Script* and for each one of these tasks, a different *DDL Model* is generated as output. The names and path files would be a combination of the name and path of the actual resource and *DDL Script*, such as *ddlFile1_Inject_DDL* for the task name and *ddlFile1_DDL_Model.xmi* for the *DDL Model* path.

7.5 Process Enactment

An introduction of the approach devised to implement the enactment of migration plans was given in Section 4.4. In this section, we give an in-depth explanation of the Process Interpretation model and the support for the execution of tasks, which is based on Trac and Mylyn.

7.5.1 Process Interpretation model

The enacting of a Concrete Migration model is itself a process that can be defined using a model that conforms to the Migration metamodel, since it involves work products, tools, activities and tasks. This model is called the *Process Interpretation model* and must be created by the designer, along with the Abstract Migration model. However, it is worth remarking that the Process Interpretation model will rarely change between migration projects, so frequently no extra effort will be needed.

The definition of a Process Interpretation model is based on the process shown in Figure 4.2, which represents the steps to be always performed by the Models4Migration tool. This process may be extended by the designer with new steps depending on the specific pair of source and target technologies, as explained later. Figure 7.7 expresses the Process Interpretation model for the process of the Figure 4.2 with MigrationDSL.

Models4Migration executes the Process Interpretation model using the Process Interpreter and returns at least two outputs, according to three mandatory activities shown in Figure 7.7: an *Automated Task file* and a *Task model.* The former is an Ant file that executes all the automated and semi-automated tasks of the migration process; it also declares every tool needed for the migration.

The Task model conforms to the *Task metamodel*, which defines tasks with names, descriptions and associated contexts (i.e. work products related to tasks). Figure 7.8 shows an excerpt of this metamodel. The Task model is an intermediate model that makes the transformation of tasks defined in the Concrete Migration model to Trac tickets easier. Note that tasks may depend on other tasks and their contexts are composed of a hierarchy of components such as projects, folders, packages and classes; this information is used by the Models4Migration tool to generate Mylyn tasks and contexts, as explained later in this section. The automated and semi-automated tasks defined in the Task model refer to the Ant tasks in the Automated Task file and their execution is performed by the Trac server.

As shown in Figure 7.7, both the Automated Task file and the Task model are generated by model transformations that have the Concrete Migration model as their input. A Process Interpretation model must always define the following three activities (along with their related tasks) to ensure the generation of those artefacts:

1. *Concrete Migration Model Generation.* It executes the *Abstract2Concrete* task, an M2M transformation which produces a Concrete Migration model from an Abstract Migration model using an Inventory model. This transformation performs the instantiation described in Section 7.4.1.

2. *Tasks Model Generation.* It executes the *Concrete2Tasks* task, an M2M transformation which produces the Task model.

3. *Automated Task File Generation.* It executes the *GenerateAutomatedTasks* task, an M2T transformation which produces the Automated Task file.

Moreover, the designer may add custom steps to the Process Interpretation model. For instance, the `MigrationGuide` elements defined in the Abstract Migration model may require an additional model, e.g. a model representing the legacy application to migrate from which to extract information for the instantiation process. The injection of such model from the legacy application would have to be performed before the *Abstract2Concrete* transformation, where the `MigrationGuide` elements are used, so an additional step would be added to the Process Interpretation model to perform such injection.

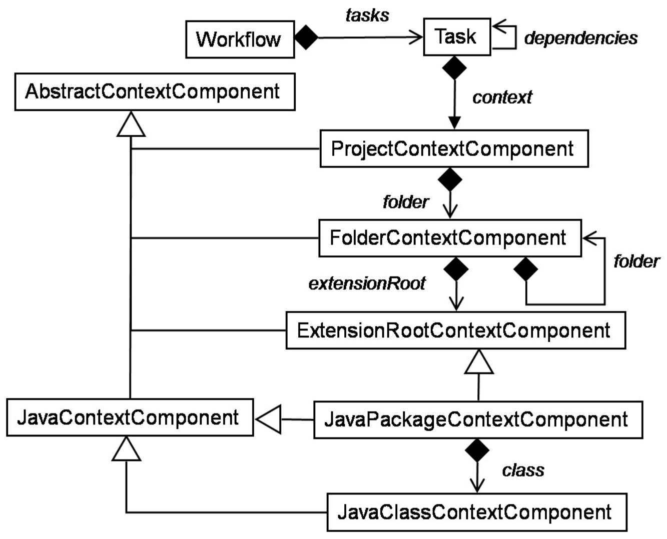


Figure 7.8: Task Metamodel.

Migration plans are enacted by Models4Migration via the execution of automated tasks and the support for manual tasks. Once the activities of the Process Interpretation model have been performed, Models4Migration exports the tasks from the Task model to a Trac server as tickets. Initially all tickets corresponding to automated tasks, which do not have any unresolved dependencies, are executed by Ant. Every time a task is completed, Trac checks if there is any automated task ready to be executed and, if so, invokes its related Ant tasks. This behaviour is a customisation of Trac that we have implemented as an extension of the *MasterTickets* Trac plugin [1].

7.5.2 Trac customisation

Trac is a bug repository aimed at creating *tickets*, which represent pending tasks to be performed and include metadata to represent information such as task description, keywords, assigned developer, etc. as well as attached files. Each ticket has a state (*new*, *closed*, *open*, *assigned*, etc.) and the transitions between states are declared in a Trac workflow. We have implemented a customisation of Trac in order to support the migration processes managed by Models4Migration.

[1]http://trac-hacks.org/wiki/MasterTicketsPlugin

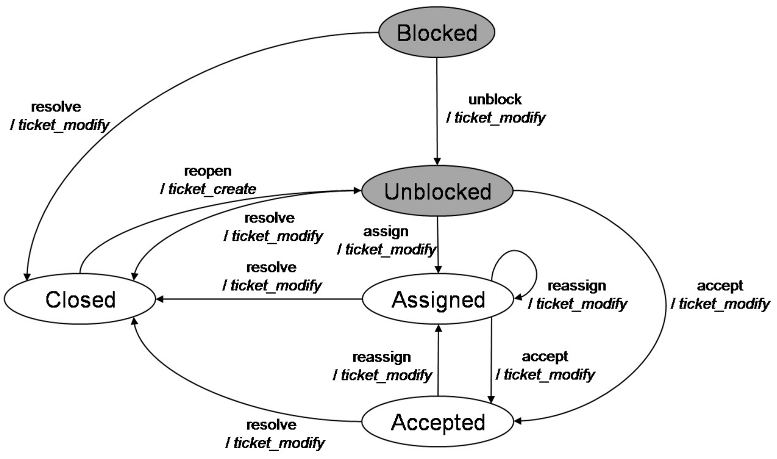


Figure 7.9: Trac transition diagram.

Tickets are used in our approach as tasks to be completed. To do this, we have defined two new states: *blocked* and *unblocked*, which are used to reflect the dependency relationships defined by the activities and tasks of the concrete plan. A blocked ticket represents a task whose dependencies have not been satisfied, i.e. the tasks (tickets) that are blocking it have not been completed yet. An unblocked ticket is a task without dependencies or whose dependencies have been satisfied and is available for execution.

Figure 7.9 shows the Trac transition diagram (i.e. the Trac workflow) for a migration process. It can be seen the different states of the tickets and the transitions among them (gray for new states and white for original states). Each transition is defined by an event and an associated action (shown in italic style). For example, the transition between the *Blocked* and *Closed* states is triggered by a *resolve* event, and it executes the *ticket_modify* action to mark the ticket as closed.

A customisation of the MasterTickets Trac plugin has been implemented in order to support the blocked and unblocked states. This plugin allows specifying dependency relationships among tickets, but it does not manage the state changes concerning these dependencies. The developed plugin extension provides automatic blocking and unblocking of tasks. Whenever a ticket is completed, all the dependant tickets are checked and unblocked if necessary.

Tickets are assigned to developers according to the roles (`RoleUse`) to which the manual and semi-automated tasks are related. Since Trac does not support users, the tickets are assigned by managers to the users (developers) defined in the server on which Trac runs, so a server software (e.g. Apache server) must be configured with the same roles that appear in the Abstract Migration model.

7.5.3 Mylyn integration

Mylyn is an Eclipse plugin that manages bug repositories such as Trac or Bugzilla, among others. An essential feature of Mylyn is the ability to establish task *contexts*. A context is the set of resources that are involved in a certain task. Mylyn obtains and stores the task contexts in the form of files attached to tasks. Whenever a context is loaded by Mylyn, the Eclipse perspective shows only the resources declared in that context. This makes the developers focus on the tasks they are assigned, increasing their productivity [128]. In addition, Mylyn supports querying the Trac task repository and updating the state of the tasks. In the Models4Migration tool, the integration between Mylyn and Trac is used in the following way:

1. The context of a task (i.e. its related work products) is encoded as an XML file and attached to the corresponding ticket in the Trac repository. This context is defined in the Task model using the classes that extend `AbstractContextComponent` (see Figure 7.8). This file is a Mylyn context.

2. The context of a task is obtained and interpreted by Mylyn to configure the visualisation of the Eclipse IDE, displaying only the related work products.

3. A Mylyn query is configured by Models4Migration to retrieve only unblocked tickets from the Trac server, so developers can only work on the available tasks, i.e. the tasks that have all their dependencies satisfied.

4. A task can be marked as completed within the user interface of Mylyn. A trigger in the MasterTickets Trac plugin customisation checks the dependant tasks and unblock them, being the changes automatically visible in Mylyn. Moreover, this trigger detects which of this dependant tasks are automated tasks and executes them by invoking their related Ant tasks.

7.5.4 Enactment of the running example

Following with the running example, the Concrete Migration model derived from the Abstract Migration model shown in Figure 7.5 is transformed into an Ant file and a Task model. The Ant file defines an Ant task for all the automated tasks defined in the Concrete Migration model, so these tasks can be accomplished just by executing the suitable Ant tasks. The Task model includes all the tasks of the Concrete Migration model, which is managed by the Trac server.

The tool first creates an Eclipse project to store the result of the enactment. Then, it creates the Tickets into Trac by means of an access API, and finally it delegates to Trac the enactment of the process.

Trac will invoke the corresponding Ant task for each automated or semi-automated task that it has to be dealt with. For each manual task to be completed, Trac provides Mylyn with useful information to complete the tasks. In the example there is one semi-automated task is defined: the confirmation of errors detected into the legacy schema. The wizard associated to this semi-automated task is firstly executed by means of an invokation to an Ant task. The control flow is stopped while the user is in charge of selecting the errors in the schema to be fixed. When the user confirms the errors, the wizard application automatically generates a fixed data model that will be the input of an automated task, and the control flow is unblocked. In the end, all the automatically generated and manually coded files are included in the Eclipse project.

7.6 Use of the Models4Migration tool

This section covers the use of the Models4Migration tool, describing the creation of Migration Cartridges and the enactment of the processes encapsulated in such cartridges. Afterwards, the migration from Oracle Forms to Java is used as a real-world case study of the use of our tool, showing the results of its application. As introduced in Section 2.5, Models4Migration distinguishes between three user roles: the *designer* of a migration plan, the *manager* of the migration process and the *developer* that performs the manual and semi-automated tasks. Figure 7.10 shows the three components provided for supporting the enactment (*Process Interpreter*,

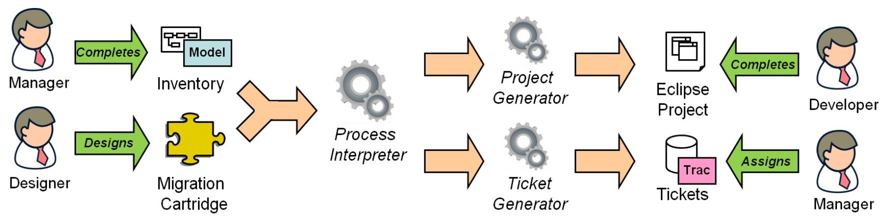


Figure 7.10: Roles using the migration tool.

Project Generator and *Ticket Generator*), as well as the interactions between roles and our tool, which are explained in greater detail below.

7.6.1 Creation of Migration Cartridges

When a migration project between a pair of specific source and target technologies is started by a company, one or more designers should define a Migration Cartridge (see Section 4.4.4) containing: i) an Abstract Migration model, ii) a Process Interpretation model and iii) the executable artefacts and resources referred by the Abstract Migration model. For each task in the Abstract Migration model, the designers must specify the input and output work products, as well as the tools used to accomplish it.

The Process Interpretation model is created with the same tools as the Abstract Migration model (see Figure 7.7), as they both conform to the Migration metamodel; however, the Process Interpretation model must always define three activities that are mandatory for the enactment of processes, as explained in Section 7.5.1, along with their related tasks, work products and tools. Moreover, the designer can add additional activities, tasks, work products and tools to the Process Interpretation model. The executable artefacts and tools that support the execution of automated and semi-automated tasks, such as RubyTL M2M transformations or Gra2MoL injections, must also be defined in the Abstract Migration model and should be included in the Migration Cartridge for the sake of encapsulation and ease of use.

Finally, the Migration Cartridge is plugged into the Models4Migration tool by simply deploying it (i.e. copying it to the file system). The Migration Cartridge for the running example is composed of the Abstract Migration model shown in

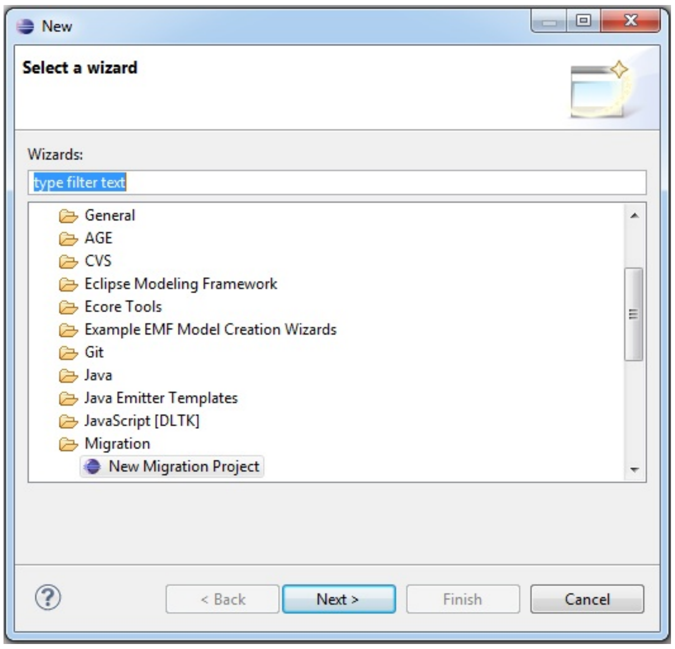


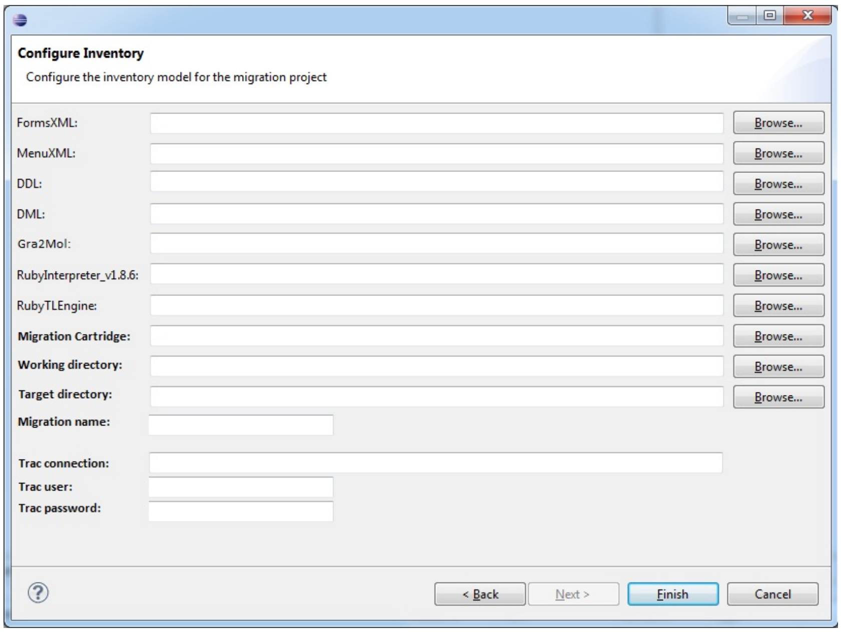
Figure 7.11: Creating a migration project.

Figure 7.5, the Process Interpretation model shown in Figure 7.7 and the resources required by the former (e.g. the *JPA.ecore* metamodel and the *GenerateJPA.m2t* template which support the *GenerateJPA* task defined in the abstract plan).

7.6.2 Enactment of Migration Cartridges

In order to enact a migration plan, the Migration Cartridge that represents it must be enacted, meaning that its Process Interpretation model must be interpreted by the Process Interpreter and the resulting tasks must be performed. This is done by one or more managers and developers. The enactment process also requires an Inventory model which declares the input parameters of the Process Interpretation model and the project deployment paths (e.g. working and generation directory), as explained in Section 7.4.2.

Firstly, the manager of the migration creates a new Eclipse Migration Project by means of Models4Migration as shown in Figure 7.11. Then, a wizard that automatically generates an Inventory model is shown to the manager to introduce all the information required to generate such model. Figure 7.12 shows an example


Configure Inventory

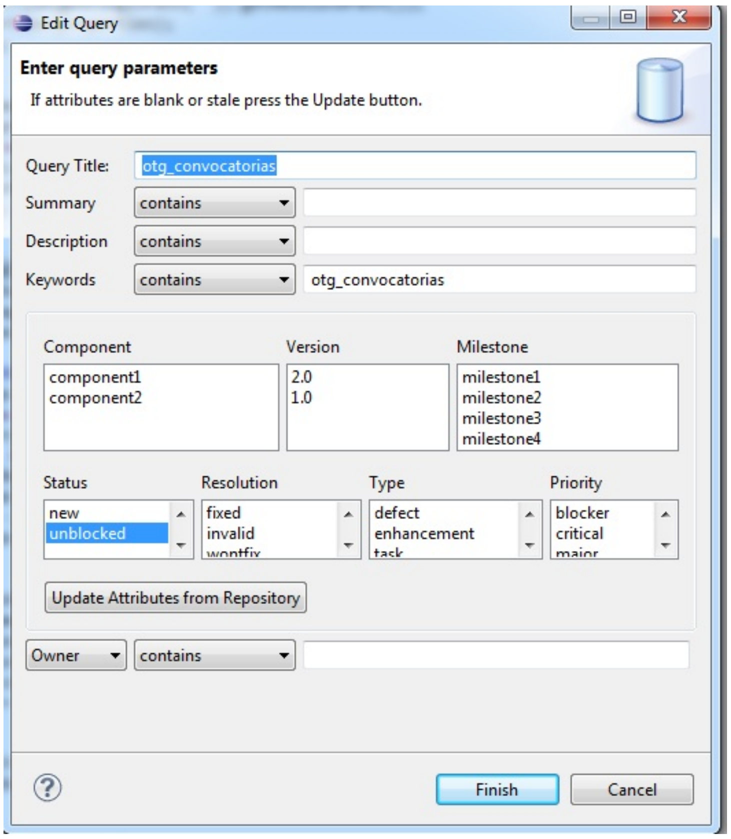
Configure the inventory model for the migration project

FormsXML:
MenuXML:
DDL:
DML:
Gra2Mol:
RubyInterpreter_v1.8.6:
RubyTLEngine:
Migration Cartridge:
Working directory:
Target directory:
Migration name:
Trac connection:
Trac user:
Trac password:

Figure 7.12: Introducing data into the Inventory model.

of such a wizard. The information can be migration-dependant, e.g. paths to the initial work products and tools, which is gathered by the plain-labelled fields shown at the top of Figure 7.12, or migration-independent, e.g. parameters needed for the tool execution (see Section 7.4.2), which are gathered by the bold-labelled fields shown at the bottom of Figure 7.12. Once these fields are filled, the wizard automatically generates the Inventory model.

Then, the Process Interpreter is executed, which creates a Concrete Migration model, and based on this model it creates the Ant file with the automated tasks, and the Task model which contains all the tasks. Next, the Project Generator creates a new Eclipse project in which all the automatically generated files will be placed. The Ticket Generator analyses the Task model and exports the tasks to a Tract server as tickets. If there are any unblocked automated tasks, they are executed. The manager is then shown a set of Mylyn tasks representing the manual and semi-automated tasks that are unblocked, that is, ready to be performed; each one of these tasks must be assigned by the manager to one or more developers. The manager can query Mylyn from the Eclipse enviroment to obtain all the unblocked


Edit Query

Enter query parameters

If attributes are blank or stale press the Update button.

Query Title:	otg_convocatorias	
Summary	contains	
Description	contains	
Keywords	contains	otg_convocatorias

Component	Version	Milestone
component1	2.0	milestone1
component2	1.0	milestone2
		milestone3
		milestone4

Status	Resolution	Type	Priority
new	fixed	defect	blocker
unblocked	invalid	enhancement	critical
	wontfix	task	major

Update Attributes from Repository

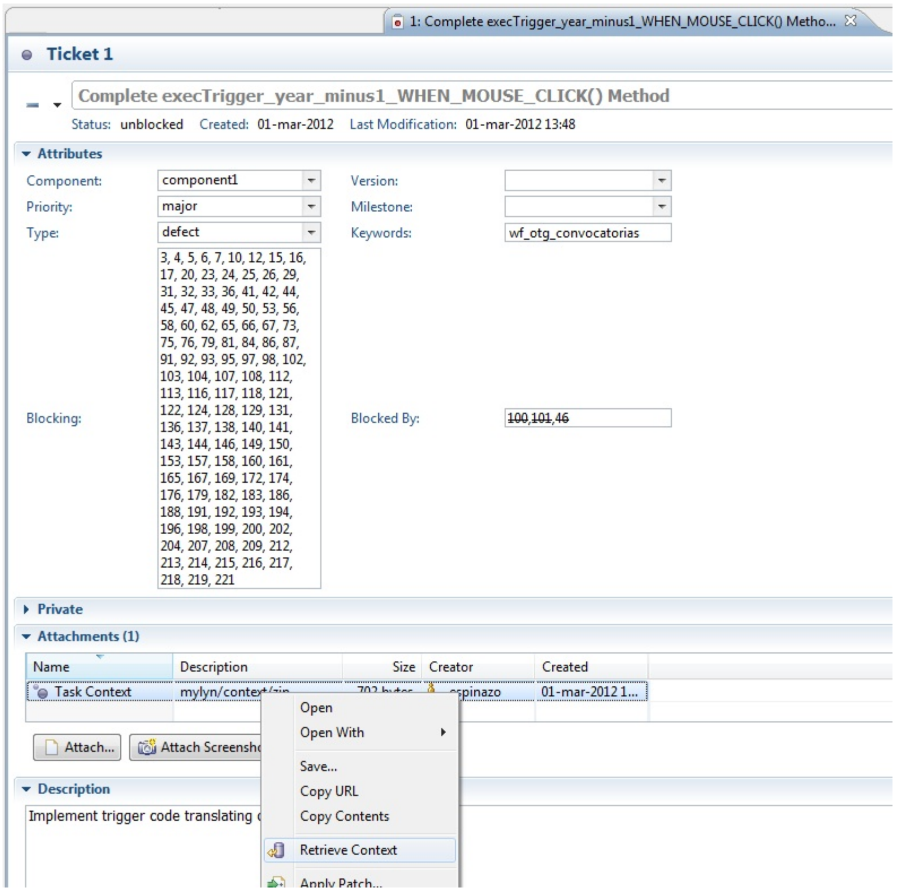
Owner	contains	

Finish Cancel

Figure 7.13: Mylyn can query and obtain the Trac tickets.

tasks by using the name of the migration project as specified in the Inventory model (this is shown in Figure 7.13). When a manager selects an unblocked task from the query, some information about the task is shown, e.g. the description and the dependant (blocking) tasks, as shown in Figure 7.14. Note that the task context is retrieved using an option of the context menu.

Finally, the developer that is assigned a task is shown only the resources that are necessary for that task. Figure 7.15 shows the perspective of a developer for the manual migration of a Forms trigger: the Eclipse environment is focused on the Java method corresponding to the trigger, showing only the resources related to the completion of the task. The code of the method also includes some guiding information as code comments about what to do, making the task easier. Note that neither the managers nor the developers are aware of the generation of the



1: Complete execTrigger_year_minus1_WHEN_MOUSE_CLICK() Metho...

Ticket 1

Complete execTrigger_year_minus1_WHEN_MOUSE_CLICK() Method

Status: unblocked Created: 01-mar-2012 Last Modification: 01-mar-2012 13:48

Attributes

Component:	component1	Version:	
Priority:	major	Milestone:	
Type:	defect	Keywords:	wf_otg_convocatorias
Blocking:	3, 4, 5, 6, 7, 10, 12, 15, 16, 17, 20, 23, 24, 25, 26, 29, 31, 32, 33, 36, 41, 42, 44, 45, 47, 48, 49, 50, 53, 56, 58, 60, 62, 65, 66, 67, 73, 75, 76, 79, 81, 84, 86, 87, 91, 92, 93, 95, 97, 98, 102, 103, 104, 107, 108, 112, 113, 116, 117, 118, 121, 122, 124, 128, 129, 131, 136, 137, 138, 140, 141, 143, 144, 146, 149, 150, 153, 157, 158, 160, 161, 165, 167, 169, 172, 174, 176, 179, 182, 183, 186, 188, 191, 192, 193, 194, 196, 198, 199, 200, 202, 204, 207, 208, 209, 212, 213, 214, 215, 216, 217, 218, 219, 221	Blocked By:	~~100~~,~~101~~,46

Private

Attachments (1)

Name	Description	Size	Creator	Created
Task Context	mylyn/context/zip	702 bytes	espinazo	01-mar-2012 1...

Description

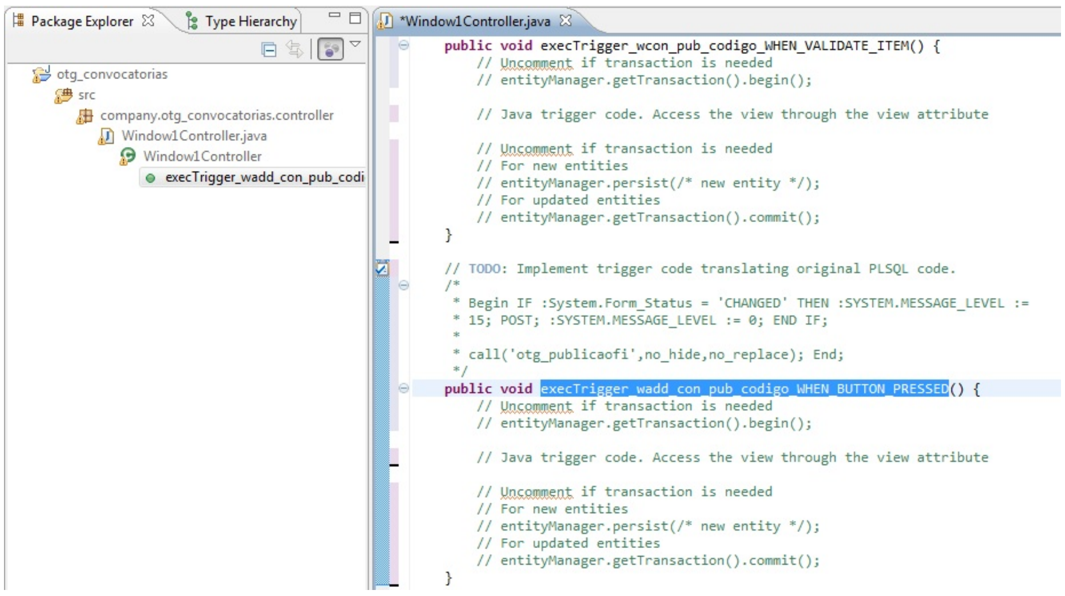
Implement trigger code translating

Figure 7.14: Ticket information.

Concrete Migration model, since they only have to deal with Mylyn tasks and their associated contexts.

7.7 Applying our tool to the real-world case study

Models4Migration has been applied in a pilot project aimed to apply MDE techniques in software migration, as commented in Section 7.2. The goal of the case study was to migrate a research project management application from the *Oracle Forms 6i* platform to the Java platform, specifically based on the *Swing* and *JPA* technologies. The migration has been defined as a software process that integrates automated tasks (e.g. obtaining Java GUIs from Forms and generating the per-



```java
    public void execTrigger_wcon_pub_codigo_WHEN_VALIDATE_ITEM() {
        // Uncomment if transaction is needed
        // entityManager.getTransaction().begin();

        // Java trigger code. Access the view through the view attribute

        // Uncomment if transaction is needed
        // For new entities
        // entityManager.persist(/* new entity */);
        // For updated entities
        // entityManager.getTransaction().commit();
    }

    // TODO: Implement trigger code translating original PLSQL code.
    /*
     * Begin IF :System.Form_Status = 'CHANGED' THEN :SYSTEM.MESSAGE_LEVEL :=
     * 15; POST; :SYSTEM.MESSAGE_LEVEL := 0; END IF;
     *
     * call('otg_publicaofi',no_hide,no_replace); End;
     */
    public void execTrigger_wadd_con_pub_codigo_WHEN_BUTTON_PRESSED() {
        // Uncomment if transaction is needed
        // entityManager.getTransaction().begin();

        // Java trigger code. Access the view through the view attribute

        // Uncomment if transaction is needed
        // For new entities
        // entityManager.persist(/* new entity */);
        // For updated entities
        // entityManager.getTransaction().commit();
    }
```

Figure 7.15: Eclipse environment context for a manual task.

sistence layer code from DDL scripts) with tasks to be manually performed by developers (e.g. writing code for the business logic layer).

7.7.1 Context of the case study

In this subsection we will present the technologies involved in the case study as well as the resources which compose the application to be migrated.

Technologies Oracle Forms 6i is a discontinued technology for developing 2-layers applications, where all the code of a system is in the client side and just database is in the server side. Swing and JPA are standard technologies provided by the Java platform for implementing the user interface and persistence layers, respectively.

Resources of the legacy application The legacy application to be migrated is a software system for managing the lifecycle of research projects: project register according to a call, budget distribution and outgoings, among others. The source application contains 28 Forms resources such as FMB, MMB and PLL files [2]; moreover, it uses an Oracle database schema with 112 tables and other database resources such as views or constraints.

7.7.2 Evaluation of the approach

We have conducted an experiment with 7 programmers and 2 analysts to validate our approach. They are workers from the company that proposed our case study. The programmers have played the role of migration developers and the analysts have acted as migration designers and managers. The designer role requires some skill of modelling because they should define the Abstract Migration plan by means of the provided DSL. The manager role should have proven specific abilities in project management owing to it must assign manual tasks to developers and monitor the status of the migration project. The developer role does not require concrete skills, apart from being able to complete the manual migration tasks which usually imply to implement new code.

Methodology The methodology we have used is the following. Firstly, in order to use our tool, a Migration Cartridge (namely *Forms2Java*) was created, containing: i) an Abstract Migration model which defines how to migrate Oracle Forms 6i applications to the Java platform using MDE techniques, ii) a Process Interpretation model similar to the one shown in Figure 7.7, and iii) all the resources required for the execution of the tasks defined in the Abstract Migration model.

Designers have defined both the Abstract Migration and Process Interpretation models. Migration developers have provided all the resources referred in the Abstract Migration.

Needless to say, the project had to be deployed in an execution environment which provided all the tools to run the Models4Migration tool and enact the migration process, such as model transformation engines (e.g. the RubyTL transformation engine).

The defined Abstract Migration model contains over one hundred elements:

[2]Forms window files, Forms menu files and Forms libraries, respectively.

- 72 work products such as a DDL script, 15 Ecore models, 15 Ecore metamodels, 11 M2M transformations, 2 T2M transformations, 7 M2T transformations, and 8 Java classes with 26 methods that represent the generated components of an MVC architecture (e.g. windows, widgets, controllers and data elements).

- The activities and tasks for injecting Forms models and transforming them into intermediate GUI models that are finally transformed into Swing source code. These activities include the injection of the Forms model, the generation of the view and controller layers using the GUIZMO framework [9], the migration of the database as explained in the running example and the manual completion of each layer.

- 34 tasks that implement the activities.

Results After applying Models4Migration, a Concrete Migration model was obtained which defines, among others, 73 Java classes with over 600 methods. Moreover, more than 800 tasks were generated, from which about 60% were manual tasks, 35% automated tasks and 5% were semi-automated tasks (see Figure 7.16).

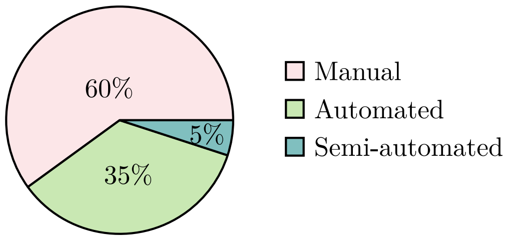


Figure 7.16: Tasks generated for the case study.

As specified in the Abstract Migration model, the persistence layer was completely generated as JPA code and the presentation layer was completed up to an 80% (some customisation is needed after the generation). Therefore, most manual tasks were related to the implementation of the domain layer (i.e. business logic). Only a 10% of business code was generated, which corresponds to method definitions and comments. Figures 7.17 depicts the percentage of code generated automatically by our proposal (*automatic*) and code which required to be implemented manually (*manual*).

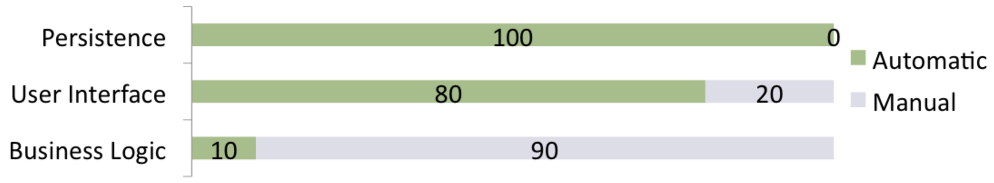


Figure 7.17: Percentage of code generated in the application layers.

Assessment by the migration participant Our tool provided a useful planning (in terms of sequenced tasks) to guide the migration project, so the tasks can be easily managed and distributed among the developers involved in the migration project by using the generated manual Mylyn tasks. This plan was strongly well valued by all the participants included in the migration owing to the next reasons:

- Designers saved time and effort in creating tickets as they only had to assign tasks to developers.
- Managers were freed from the task of creating of the Eclipse project. Moreover, they did not have to deal with automated ANT scripts, because these scripts were automatically generated, and the code was error-free and the scripts were ready to be executed.
- Programmers were assisted in completing manual tasks by means of providing the fragment of legacy code to be migrated in each task (inside comments) and the Mylyn context which helped developers to focus on the specific resources (usually classes) involved in the task.

Assessment in terms of productivity An estimation of the productivity gains of our approach for a concrete migration project requires comparing the cost of creating the Abstract Migration model and the benefit in automatically generating the artefacts obtained as output of the enactment: automated task files, the Trac tickets and the Eclipse configured project. In our case, creating the Abstract Migration model with our MigrationDSL required 4 hours. We estimated in 80 hours the manual creation of: i) the ANT file with the automated tasks (18 hours = 280 automated tasks, 4 min. per task definition), ii) the Mylyn tickets with the manual and semi-automated tasks (52 hours = 520 manual and semi-automated tasks, 6 min. per task definition) and iii) the Eclipse project with the skeleton of the new classes (10 hours), which are generated by our tool.

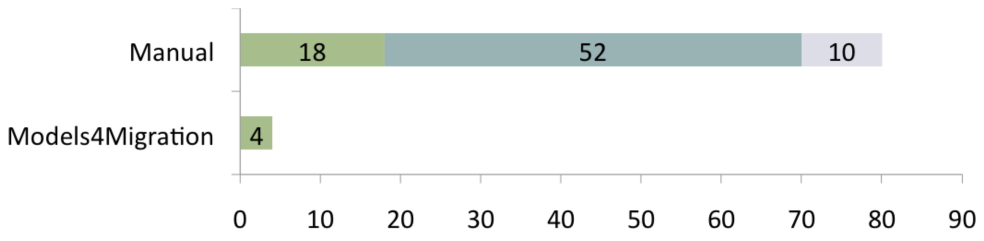


Figure 7.18: Hours to create the tasks of the migration plan.

The Models4Migration implementation cost (including the MigrationDSL definition) took about 360 hours (2 developers dedicating about 30 hours each one per week during 6 weeks). Therefore, the number of migration projects needed for the return of investment (ROI) is 5, assuming our case study as a medium-sized project.

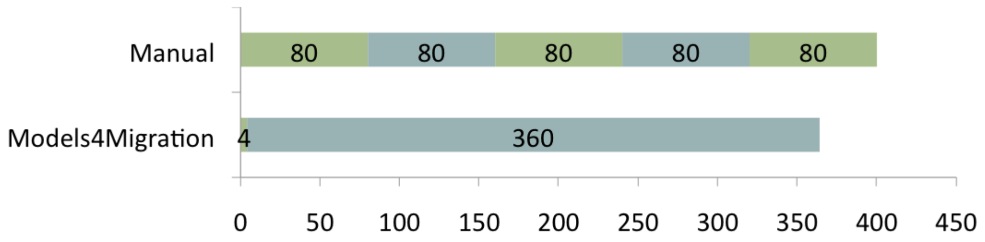


Figure 7.19: Hours for the ROI of our approach.

7.8 Lessons learned

Since that one of the contributions of our work is applying MDE techniques to implement a process engineer tool, in this section we present some of the benefits and drawbacks of MDE that we have obtained during the implementation of the tool, as well as the qualities that MDE has brought to out solution.

- *Productivity.* This is one of the most important benefits when using MDE. The use of transformations provides automation that results in productivity. The development of a model-based solution requires the effort of designing the metamodels and transformations, which is later compensated by the amount or size of the artefacts that are automatically generated. As a rule, an MDE solution is more profitable as it is applied to a greater number of source artefacts. In our case, transformation chains allow us to automate the instantiation and enactment phases. In implementing process engineering tools, the usage of MDE techniques seems more convenient than traditional techniques (e.g. general-purpose programming languages) as the management of process models is the central task. For instance, model transformation languages allow writing code in a declarative style. We have used the RubyTL language whose hybrid nature [129] eases to implement complex transformations. In addition, the RubyTL modularity mechanism helped us to write composable and reusable transformations as explained in [129].

- *Technology independence.* Our approach is independent of the project management tools (i.e. Ant, Trac and Mylyn) that we use to create the final system. This independence is achieved thanks to the Task metamodel that decouples the migration definition and instantiation from the enactment with the mentioned management tools. Task models express tasks without relying on technology-specific concepts. Therefore, if we would like to use different management tools, only the model-to-text transformations based on the Task model would need to be replaced.

- *Reusability.* Metamodels and transformations can be reused between different projects, thus reducing the effort of developing MDE solutions. We have reused standard metamodels, particularly the SPEM and the UML2 Activity Diagram metamodels, what greatly simplified the creation of the Migration metamodel as we only had to add some concepts on top of these specifications. Transformations can be reused by defining metamodels that make some parts independent of other parts and defining the transformations from and to those metamodels. This mechanism can be used to reuse metamodels and transformations. In our case, we had a metamodel for representing tasks (the Task metamodel) and a Java program that takes Task models and interacts with the Trac server, so we were able to reuse that metamodel and the Java program in our solution just by transforming from the Concrete Migration model to the Task model.

- *Extensibility.* A transformation chain can be extended with new stages or even it can be modified with new stages that replace the existing ones. Therefore, models can be seen as extension points for modifying or adding new stages to a chain. As we already mentioned in Section 7.4, new steps could be added to the Process Interpretation model if it were required, for example injecting a source code model that would be used in the *Abstract2Concrete* transformation. This feature of MDE endows our solution with flexibility to be used in projects of different nature.

- *Easy creation of DSLs.*

  As noted in [130], DSLs have been a part of the computing landscape since the early years of programming. However, the emergence of MDE has originated a growing interest in DSLs and the development of metamodelling-based tools (language workbenchs) supporting the definition of textual or graphical DSLs. In our case, we tackled the development of the graphical DSL after the metamodel concepts were estabilised as it is recommended in [131].

- *Problems to work with big models.* The instantiation of the Abstract Migration model in a complete real migration can result in thousands of elements in the Concrete Migration model. Moreover, the injection of the models specified in the Abstract Migration model can be slow and the tools can even crash because the system runs out of memory. The management of big models is a current research topic in MDE. Morsa citemorsa and CDO [114] are two of the most promising efforts in this area.

- *Immaturity of the MDE development environments.* Despite being useful, MDE lacks of mature development environments, which limits its applicability in industrial settings. For example, debugging model-to-model transformation is more complicated than in common programming languages, since no interactive debugging or high level information is displayed. Transformation testing is another example. Although there are works aimed at addressing these problems, the solutions are neither robust nor integrated in MDE tools.

7.9 Conclusions

The tool presented in this chapter addresses the requirements mentioned in Section 2.5. It is agnostic to any specific technology, hence achieving a relevant degree of flexibility and reusability. We assume that the source artefacts of the application to migrate are available to the tool. If this assumption was not true the approach that we propose would still be valid, but activities would have to be included in order to recover or infer information about the source from the execution traces. It is worth noting that the definition of a migration plan actually requires more complex activities than defining the development task workflow, such as viability analysis, cost evaluation or version control, which are out of the scope of this work.

When addressing the definition of the proposed reengineering approach we realised the convenience of having a tooling to support model-driven migration processes. We have therefore tackled the building of a tool named Models4Migration for supporting the definition and enactment of model-driven migration processes. Four basic functionalities have been identified for this tool: i) provide a means to specify migration workflows (i.e. model transformation chains); ii) ability to execute the automated tasks (e.g. model transformations) without human intervention; iii) support and guidance for the development of manual tasks, and iv) integration of manual and automated tasks into the migration workflow.

Since the SPEM extensions support the enactment of software development processes, they are not appropriate to our goal, so we have defined our own DSL named MigrationDSL. While SPEM extensions are applicable to any software process, our language is tailored to model-driven migration processes. This allows us to achieve the execution of automated tasks and the support for manual programming tasks through the integration of the Trac and Mylyn tools. Thus, the developers involved in a migration project are guided based on the task context and team leaders save time thanks to the automatic generation of tickets (i.e. Mylyn tasks). It is worth noting that this support for manual tasks which is based on the integration of Trac and Mylyn is one of the novel contributions of this work.

It is clear that our proposal will produce important benefits when, due to the nature of the migration project, the majority of the generated tasks are automatic. However, even though the migration scenario produces a number of manual tasks much bigger than the automatic ones the tool is still useful. In this case, the migration managers remark that our tool provides a real ticket management support by Mylyn and Trac. They find very useful that all the manual tasks are described by tickets which offer a project environment and a set of application resources defining a context. Moreover, the tasks are related which imposes the completion of some tasks before finishing other ones.

The use of MDE has been crucial for achieving the goals, since the implementation of the functionality of the tool has been tackled using metamodels, models and model transformations. To our knowledge this work is the first one that describes in detail how MDE techniques can be applied to building a software production environment. A chain of model transformations generates the artefacts that are needed (i.e. Ant file and Trac tickets) to enact a migration process model expressed with the MigrationDSL language.

The approach is technology-independent, which means that it can be configured to deal with specific technologies by means of the cartridges, which encapsulate the migration strategy (i.e. the Abstract Migration process model) and their interpretation (i.e. a Process Interpretation model). Once a cartridge is plugged into Models4Migration, new migration projects can be created in Eclipse from an Inventory model that specifies the involved artefacts.

Finally, we have shown how the different stakeholders involved in a migration process can use our tool and we have provided some information related to its application in a real case study.

”Should I bolt every time I get that feeling in my gut when I meet someone new? Well, I’ve been listening to my gut since I was 14 years old, and frankly speaking, I’ve come to the conclusion that my guts have shit for brains.”

from High Fidelity
(Suggested by Fernando)

8 Tool Interoperability

Software reuse is one of the techniques used to insure quality and productivity in the building of new solutions. As we showed in Chapter 6, the data reengineering approach proposed is required to integrate the Concept Explorer (ConExp) tool into the normalisation step in order to identify functional dependencies. We then experimented in building an MDI bridge for this tool. This bridge is in fact solely required to implement a syntax mapping for ConExp (i.e. injector and extractor) because its output data were injected into FD models that are used in our approach. From this experience, we continued exploring the MDE capabilities to integrate tools. In particular, we first build the infrastructure (i.e. metamodel, injectors and extractors) so as to integrate the DB-Main tool into MDE setting, and we then used this tooling to create a bidirectional bridge between the DB-Main and Objectiver tools. More specifically, the objective of this bridge is to interchange DB-Main database schemas and Objectiver Object models. The integration of DB-Main into MDE allowed us to create several injectors and extractors, and building the DB-Main/Objectiver bridge required us to address the implementation of bidirectional semantic mappings.

The basis of the Model-driven interoperability we stated in Section 2.4. A detailed description of the architecture of the Objectiver/DB-Main bridge will be presented in the following sections. We will start by presenting the tools involved in the integration and shall then define the pivot metamodels used in this kind of solution. After that, we shall describe how we have implemented the syntax and semantic mapping, which bridge the gap between the two tools. The former deals with aspects related to the data format of each tool, while the latter covers the conceptual differences of the data of each tool.

8.1 The tools

Building software systems involves using multiple tools with different purposes, which cover all the stages of the software development life-cycle. Therefore, tool integration has always been a topic of great interest for the software community [50]. We have tackled the integration of the DB-Main database engineering tool [1] with the Objectiver requirement engineering tool [2], as part of a collaboration with the Precise research group, which created DB-Main at 1993.

DB-Main is a mature and free tool with a rich functionality for data engineering. To facilitate integration with other tools, DB-Main provides an API Java named JIDBM (Java Interface for DB-Main) which allows data and metadata of DB-Main projects to be manipulated. The DB-Main project files (*.lun* extension) offer an alternative to the use of this API; they represent all the information involved in a DB-Main project in a non-XML proprietary format (hereafter LUN format), which defines a complex structure with which to save and load the DB-Main projects. In addition, a plugin that exports historical data and schemas in XML format is being developed, and an initial version of this plugin is supported in a beta version (at this moment the tool does not yet support the XML importation).

KAOS is a well-known goal-oriented requirement engineering method [132]. The basic concepts of this method are the following: (i) requirements are described by

[1]http://www.db-main.be/

[2]http://www.objectiver.com/

means of a hierarchy of goals (i.e. desired system properties); (ii) each goal is assigned to the agent (or agents) responsible for achieving it; (iii) goals involve domain entities (a.k.a. objects) and relationships between them; and (iv) behaviour that agents need to fulfill is expressed by means of operations, which are triggered by events, and work on objects. A requirement elicitation is therefore expressed by means of four kinds of models: Goal model, Responsibility model, Object model and Operation model. *Objectiver* is a payment tool supporting KAOS methodology: it allows analysts to create KAOS models and generates requirement documents conforming to existing standards, among other functionalities. This tool not only allows the export/import of project data in XML format, but also as Ecore models [43]. To achieve this, Objectiver defines an Ecore metamodel which describes the four kind of KAOS models. This feature promotes the MDE interoperability of the tool as will be discussed later.

We have experimented the building of an MDE interoperability bridge between Objectiver and DB-Main (henceforth *Objectiver/DB-Main*), whose architecture was illustrated in Figure 4.4. More specifically, a bridge able to establish a bidirectional mapping between Objectiver object models and DB-Main logical schemas. Objectiver object models (which represents the concepts of the application domain in KAOS methodology) are used to create a database schema in DB-Main. On the other hand, the changes applied to database schemas in DB-Main are propagated to the object model managed by Objectiver. Therefore, the solution not only allows the generation of a final system's software artifact (i.e. a database schema) from a requirement model (i.e. the object model), but the inverse transformation is also possible. In this way, the object model and the schema are kept synchronised.

8.2 Pivot Metamodels

In building a MDI bridge, the first stage is the creation of the pivot metamodels. In the case of the Objectiver/DB-Main bridge, only the pivot metamodel for DB-Main must be created. Objectiver includes a KAOS metamodel that defines 44 elements, but we have only considered those related to the object model. Figure 8.1 shows this part of the KAOS metamodel. An Objectiver model is represented by a *KModel* and this is made up of one package (*KPackage*) named *rootPackage*.

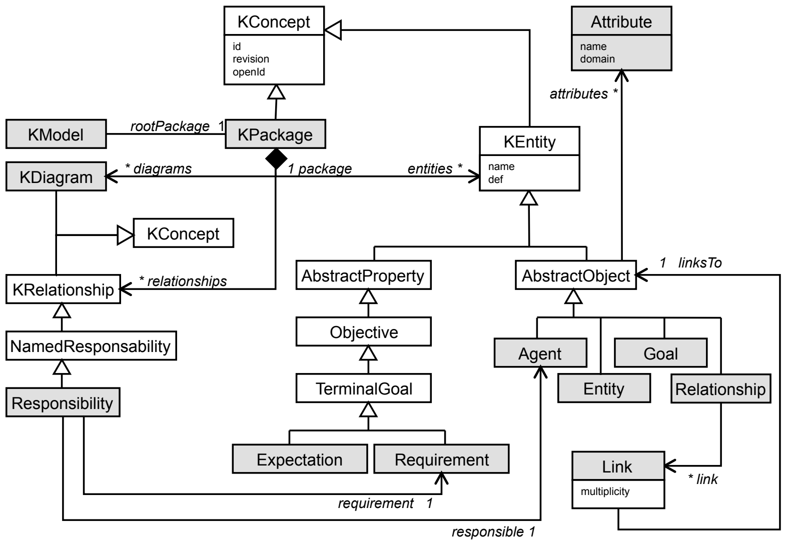


Figure 8.1: Excerpt of the Objectiver pivot metamodel.

Packages are composed of KAOS diagrams (*KDiagram*), KAOS entities (*KEntity*) and KAOS relationships (*KRelationship*). KAOS Diagrams provide graphical data in order to visualise models. KAOS Relationships allow the definition of responsibilities (*Responsibility*) between agents (*Agent*) and requirements (*Requirement*). KAOS Entities are the basic elements of the Objectiver models. In addition to the requirements and expectations properties (*Expectation*), the following abstract objects (*AbstractObject*) are defined: (i) agents (*Agent*), (ii) goals (*Goal*), (iii) entity objects (*Entity*) that represent the objects in the *Object model*, and (iv) relationships (*Relationship*) which contain links (*Link*) that define connections between abstract objects and register the multiplicity of the link. These abstract objects are characterised by attributes (*Atribute*) which contain a *name* and a *domain*.

We have defined the DB-Main metamodel shown in Figure 8.2. This metamodel has been devised according to the data structure used in DB-Main to represent database schemas. Thereby, the implementations of injectors/extractors to/from DB-Main are less complex. A database schema (*Schema*) is composed of a collection of tables (*Table*) and a collection of foreign keys (*ForeignKey*). Each table is formed by a collection of columns (*Column*), and references to collections of indexes (*Index*) and identifiers (*Identifier*). Each column has a type (*ColumnType*)

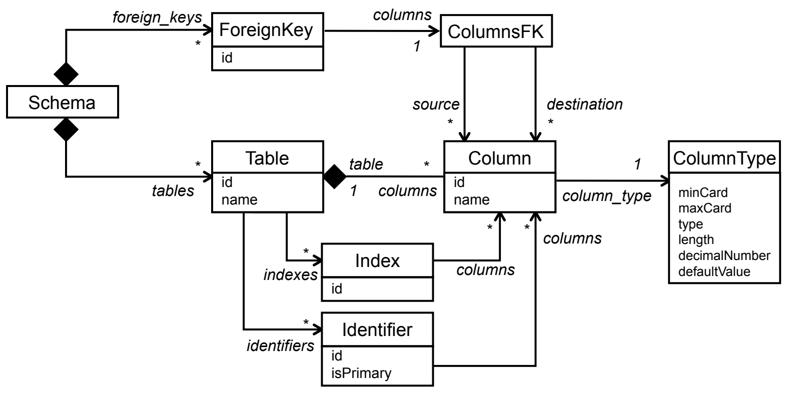


Figure 8.2: DB-Main pivot metamodel.

whose attributes are: *type* which defines the domain of a column; *minCard* and *maxCard* which define the cardinality in case of the multivalued columns; *length* which delimits the size in the case of a string type; *decimalNumber* which delimits the precision of the decimal part in the case of a number type; and *defaultValue* which establishes a default value for the type. Schemas, tables and columns have a name. Each identifier has a boolean attribute *isPrimary* to indicate if it is the primary key of the table that references it. Foreign keys (*ForeignKey*) reference the *source* and *destination* columns, which are stored in two ordered sets of the same size. Given a position, a column in the source set represents the foreign key column of a table which references a column in the destination set which represents the primary key column of another table.

8.3 Syntactic Mapping

Once the pivot metamodels are available, the injectors and extractors must be created if the tool does not support the facilities to export/import data to/from Ecore models. As indicated previously, we have only created injectors/extractors for DB-Main, since they are already included in the Objectiver tool.

Several MDE tools may be used in order to automate the creation of injectors and extractors. For XML schemas, Eclipse/EMF provides a generic injector/extractor to/from Ecore models. EMF requires the XML Schema of the XML documents, and automatically generates a metamodel which represents the information specified in the XML Schema. The EMF injector transforms a XML document conforming to the XML Schema in a model conforming to the previously generated metamodel. The EMF extractor performs the reverse process.

For a grammar format, DSL definition tools (a.k.a. DSL workbenches), such as Xtext [3] and EMFText[4], can be used to automatically generate an injector and extractor. The grammar is specified by means of the notation provided by the workbench to express the DSL grammar. Injectors can also be automatically generated by means of a text-to-model transformation language, such as Gra2MoL [47], which allows the expression of mappings between grammars and metamodels.

The use of a Java API can be automated by means of the API2MoL tool [101], which automates the creation of (i) the API metamodel, (ii) the injector that obtains a model from API objects, and (iii) the extractor that generates API objects from models.

As indicated in previously, DB-Main provides three alternatives for integrating data: the JIDBM API, and the LUN and XML formats. Figure 8.3 shows four diagrams which outline the injection/extraction strategies that can be implemented for DB-Main, which are explained below.

8.3.1 Strategies to implement the injection

For the JIDBM API (see Figure 8.3(a)), we have contrasted the manual creation of a Java application with the automated generation of the injector by means of the API2MoL tool.

- *Using the EMF API.* A Java application uses JIDBM to access the DB-Main data (it is not required that the tool was in execution) and the reflective EMF API to create the DB-Main model which represents the database schema. The persistence service of EMF is used to store the model.

- *Using API2MoL.* API2MoL automatically generates the JIDBM metamodel and the injector/extractor for this metamodel. The API2MoL input should

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

[4]http://www.emftext.org/index.php/EMFText

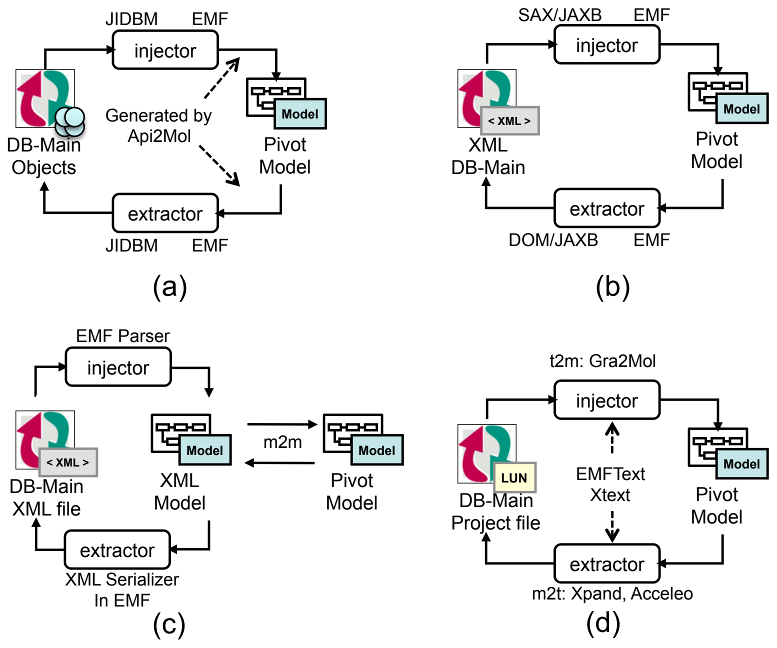


Figure 8.3: Injectors and extractors for DB-Main.

be a Java program using the API. For the bridge proposed here, the program uses JIDBM to recover all the data of a DB-Main project. Regarding the previous solution, API2MoL avoids implementing the manual task of creating the model by using the EMF API. In our case, the generated API metamodel is slightly different to the defined DB-Main metamodel. Hence an additional M2M transformation is needed, although it could also have been used as the pivot metamodel.

For the XML format (see Figures 8.3(b) and (c)), we have contrasted the manual creation of a Java application with the automated generation of the injector by means of the EMF generic injector.

- *Using the EMF API.* The EMF API can be used to create a model from either the data produced in an XML parsing or the Java objects obtained in a XML-to-Java mapping. We have created a XML parser using the SAX parser included in JAXP [5]. Although JAXP also provides a DOM parser that

[5]http://jcp.org/en/jsr/detail?id=63

loads all the data contained in the XML document in memory at once, the restrictions in the queries of the XML document promote the use of SAX. On the other hand, we have used the JAXB API [6], which maps Java objects to XML documents and vice versa. To build an injector, the *unmarshall* tool provided in JAXB allows the loading of the XML data into memory as Java objects. Then, a Java program can analyse these objects and use the EMF API to create a model that conforms to the DB-Main metamodel.

- *Using the generic XML injector provided by EMF.* As indicated above, EMF provides a generic injector for XML documents. The models injected conform to the XML Schema metamodel. An additional M2M transformation that converts the model injected into a DB-Main model would be therefore necessary.

LUN files store the data and metadata of a DB-Main project, in particular the database schema, the history of the different schema versions, and the operations applied in a data engineering process. Since the LUN format is defined by a grammar, DSL workbenches and text-to-model transformations may be used to automate the building of injectors. Therefore, we have again contrasted an EMF-based manual solution with the use of DSL workbenches, in particular EMFText, and the Gra2MoL language (see Figure 8.3(d)). However, the complexity and variability of the LUN format makes it very difficult to implement the EMFText grammar as well as the Gra2MoL transformation. For instance, the meaning of some data are dependent of their position and type in the LUN file.

8.3.2 Strategies to implement the extraction

As with injectors, we have experimented with several strategies for each DB-Main interoperability option (see Figure 8.3). In the case of JIDBM (Figure 8.3(a)), the two strategies previously described for the injectors are bidirectional because JIDBM also includes methods for creating and updating the DB-Main data. Therefore, the JIDBM and EMF APIs could be used to create an extractor manually. However, this extractor could be automatically generated by using API2MoL.

[6]http://jcp.org/en/jsr/detail?id=222

Table 8.1: Assessment of the strategies

	Strategy	Automation	Bidirectional	Easiness	Maturity
JIDBM	EMF	✘	✘	✓	✓
	API2MoL	✓✓✓	✓	✘	✘
XML	SAX/DOM or JAXB + EMF	✘	✘	✓	✓
	EMF XML + m2m	✓✓	✓	✓	✓
LUN	manual pars./gen. + EMF	✘	✘	✓	✓
	Gra2MoL + m2t	✓	✘	✘	✘
	DSL workbench	✓✓✓	✓	✓	✓

Using the XML schema, the strategies used for injectors are bidirectional, so that they can again be applied to create extractors but in the opposite direction.

- The EMF API is used to traverse the DB-Main pivot model, and a XML parser or the JAXB mapper in order to create the XML document (Figure 8.3(b)). In this scenario, the DOM parser could be more appropriate than the SAX parser due to the fact it eases the creation of XML trees.

- The generic extractor available in EMF could be used by previously writing an M2M transformation that converts the DB-Main model into a model that conforms to the XML schema's metamodel. Then, the generic extractor could serialise the XML model into a XML document (Figure 8.3(c)). It is worth remarking that a single M2M transformation could be enough if a M2M transformation language supporting bidirectionally is used.

Finally, a LUN file could be generated from a DB-Main model by using a M2T transformation (see Figure 8.3(d)). However, the complexity of the LUN grammar makes it difficult to implement this transformation and the effort required is much greater than for the other implementation strategies.

8.3.3 Assessment of the strategies

In the previous section, different strategies have been proposed which implement the Objectiver/DB-Main bridge for the EMF framework. Four main criteria could be considered to choose the most appropriate strategy:

- *Automation Level.* Instead of creating solutions from scratch by means of GPLs and APIs, MDE technology (tools and languages) can be used to automate the creation process. In this way, the effort involved in development is significantly reduced.

- *Bidirectionality.* Which bidirectionality facilities are provided should be taken into account. An ideal solution would be to implement the injection and extraction process at once.

- *Ease of learning.* Learning the applied technologies involves an effort which must be taken into consideration when evaluating the cost of developing a solution.

- *Maturity Level of Tools.* The lack of mature MDE tools is hindering its industrial adoption. Many tools stem from academic projects and they lack the required standards of quality.

Table 8.1 summarises the assessment of the applied strategies regarding the considered criteria. With regard to the level automation, the following marks are used: ✘ indicates that the solution is totally manual, ✓ if the automation is achieved by writing model transformations, ✓✓ if an additional m2m transformation is required, and ✓✓✓ when injectors and extractors are automatically generated. Bidirectionality is supported if the injector/extractor can be generated from an only specification (or program). In assessing easiness, we have considered that GPLs, involved APIs, and BNF grammars are easy to use and learn. Finally, we have considered as mature tools those created by companies or consortium that have provides an stable support for several years.

The ideal strategy would be a strategy capable of automatically creating a bidirectional solution (i.e. injector and extractor are automatically generated), and is based on mature technology which is easy to use and learn. Next, we will analyse the MDE technology involved in the outlined strategies.

API2MoL is shown as the best choice from a technical point of view, because it automatically generates a bidirectional solution. This tool also generates the API metamodel (in this case, the JIDBM metamodel), which could be adequate as a pivot metamodel. An additional M2M bidirectional transformation would be needed if the API metamodel is not the pivot metamodel, and developers should just write a simple program that uses the API for building the data objects to be injected. However, API2MoL is an academic tool that has been discontinued. Moreover, lack of documentation makes learning it difficult. It is also worth noting that the metamodel generated by API2MoL may be incomplete, as discussed in [101]. Developers should therefore write an APi2MoL specification in order to obtain the complete metamodel, as well as the corresponding injector and extractor. In our case, the metamodel generated by API2MoL was complete.

EMF also provides a bidirectional solution through the generic injector and extractor available for a XML Schema. As indicated above, an additional M2M transformation is always needed to map the XML schema metamodel and the pivot metamodel. This mapping should be implemented by means of a bidirectional transformation. Regarding API2MoL, the use of these tools is simpler and the effort involved in learning it is significant lower.

With regard to the grammar format, DSL workbenches allow the obtaining of a bidirectional solution from a BNF-like grammar specification. In addition, T2M and M2T transformations may be used to implement injectors and extractors, respectively. To our knowledge, Gra2MoL is the only available T2M transformation language, but this language is specially tailored to inject models from GPL code. A detailed comparison of Gra2MoL and DSL workbenches is discussed in [47]. On the other hand, M2T transformation languages (e.g. Acceleo and Xpand) are widely used to generate textual software artefacts from models, and they are easy to learn and use. In our case, these three strategies are the choice that involve a larger development effort due to the complexity of the LUN format.

8.4 Semantic Mapping

Once the syntactic mapping has been created, the semantic mapping could be implemented. In our case, a bidirectional mapping between the DB-Main pivot metamodel shown in Figure 8.2 and the excerpt of the Objectiver metamodel shown in Figure 8.1. A bidirectional M2M transformation should be written to implement such a mapping, instead of writing two unidirectional M2M transformations. In this way, the implementation and maintenance effort would be reduced. Among

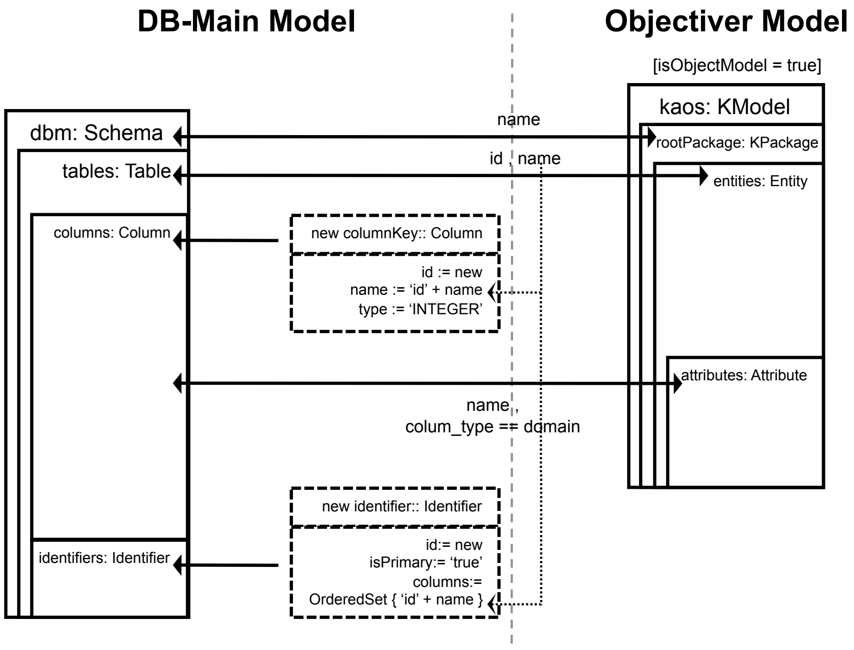


Figure 8.4: Semantic mapping for *tables* ↔ *entities*.

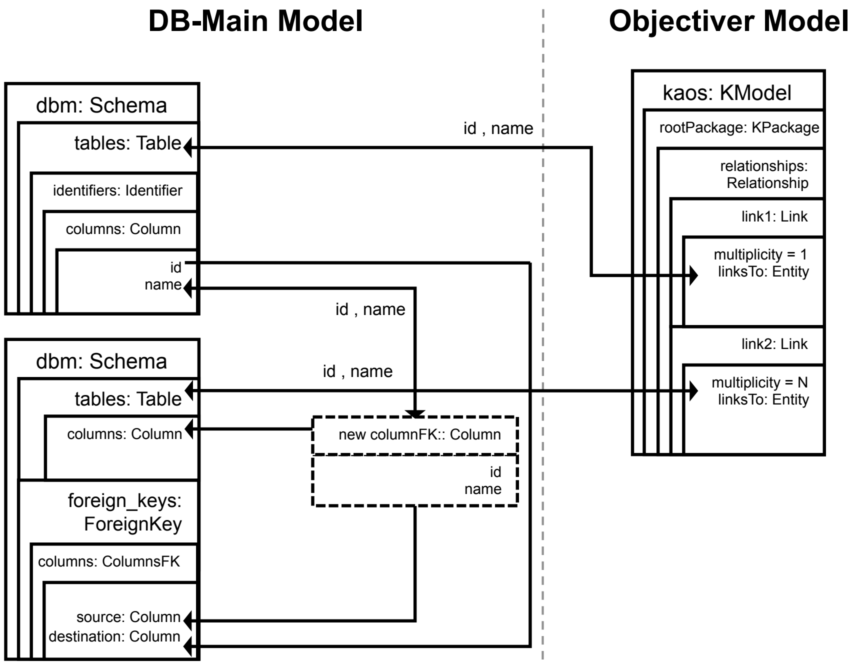


Figure 8.5: Semantic mapping for *foreign keys* ↔ *relationships*.

the most widely used M2M transformation languages, QVT Relational is the only one supporting bidirectionality. Although QVT was proposed with the purpose of becoming a standard language for M2M transformations, only a few implementations are currently available and none have achieved the desired level of maturity. ModelMorf [7] and Medini QVT [8] are the commonly used implementations. We have used Medini QVT ”due to its greater visibility on the Web, Eclipse integration, debugging facilities and other developer-friendly features” [52]. Below, we describe the defined QVT transformation.

Figure 8.4 represents the mapping applied without considering foreign keys (FK). The graphical notation depicts the metamodel’s classes as boxes and the semantic links between them as bidirectional arrows. The arrows are labeled by the attributes mapped in the link (== is used to denote the mapping but it is omitted if the name is the same in both metamodels). The reference and aggregation relationships between classes are represented by means of nested boxes. Figure 8.4 shows that a DB-Main schema (*Schema*) maps to a root package (*KPackage*) of an Objectiver model (*KModel*). The schema and the root package will have the same name, and each table (*Table*) of the schema maps to an entity object (*Entity*) of the package. A table and its mapped entity will have the same name and identifier (id). Each column (*Colum*) of a table maps to an attribute (*Attribute*) of an entity. A column and its mapped attribute will have the same name and the column type are given by the domain of the attribute. A new column for registering the primary key is directly instantiated. Its `type` is `'INTEGER'` and its `name` is formed by concatenating the table name to the prefix ’id’. It is worth noting that the attributes of an entity do not have *id* but *columns* and *identifiers* in the DB-Main model and require an unique *id*. New ids for the elements of a DB-Main table are created by systematically concatenating the *id* of the container element to a sequential counter which starts from zero for each container element. A new identifier is also instantiated. Its `isPrimary` attribute is `'true'` and its `columns` attribute is the name of the new column key added.

Figure 8.5 represents how FKs are mapped to relationships. This mapping is

[7]http://www.tcs-trddc.com

[8]http://projects.ikv.de/qvt

similar to the conversion between the Entity/Relationship model and the Relational model (without considering optional/mandatory relationships, i.e. not considering multiplicity `0`). The conversion implies that the table having multiplicity `1` propagates the columns of its primary key to the table with multiplicity `N`, where a new FK comprising the propagated columns is added. An Objectiver object model can include three kinds of relationships: `1:1`, `1:N` and `N:N`. A FK (*ForeignKey*) of a table maps a relationship (*Relationship*) between entities. A relationship has two links that represent the multiplicities at each end. For relationships `1:1` and `1:N`, each link maps a table, more specifically the table mapped to the entity referenced by the link (reference *linksTo*). In accordance with the previously explained conversion, the entity in the link with the multiplicity `1` corresponds to the table that propagates its primary column to the other table as the FK column. Note that the column propagation occurs only in the DB-Main model. As we can see in Figure 8.5, *links* are mapped to *tables* by the attribute *linksTo*. Once we have identified both the *tables* in DB-Main involved in the *relationship* of Objectiver, the propagation of the new column in the DB-Main model is simple. We should create a new column from the column that comprises the *identifier* of a table in order to represent the new *columnFK* in the other table. Both columns, the new FK column of a table and the identifier column of the other one are the source and destination columns of the *ColumnsFK* element in the DB-Main model, respectively. For `N:N` relationships a new table is generated in the DB-Main model. It is composed of the columns which comprise the primary key of the two tables involved in the `N:N` relationship. The two new `1:N` relations between the new table and the tables of the relationship are resolved by using the previous `1:N` mapping.

Once the semantic mapping of our bridge is defined, we shall illustrate how the mapping has been implemented by means of Medini QVT. We shall show only one pair of QVT relations due to space limitations.

The next relation corresponds to the mapping between schemas and Objectiver models. It is only applied when the Objectiver model is an object model. For this, the *isObjectModel()* helper checks if the name of the Objectiver model contains the ”object” string. The mapping establishes that entities and tables have the same name. Finally, the *Table2Entity* relation that maps the tables of the schema and the entities of the root package is resolved. The *TableKey2Entity* relation that establishes a table identifier for each entity is also resolved.

Listing 8.1: Medini QVT Relation for schemas and object models.

```
top relation SchemaToModel {
  n : String;
  idModel : String;
  enforce domain dbmain dbm : dbmain::Schema {
    name = n,
    tables = tb : dbmain::Table {}
  };
  enforce domain objectiver obj : kaos::KModel {
    rootPackage = root : kaos::KPackage {
      subPackages = subs : kaos::KPackage {
        name = n,
        entities = en : kaos::Entity {}
    }}
  };
  when { isObjectModel(n);}
  where {
    idModel = root.id.substringBefore(':');
    TableKey2Entity(tb,en, idModel);
    Table2Entity(tb,en, idModel);
  }
}
```

The relation that maps to $1 : N$ relationships is shown below. When this relation is applied the identifiers are mapped and the *ColumnsFK2Links* relation is resolved in order to map the FK columns and the two links of the relationship.

Listing 8.2: Medini QVT Relation for foreign keys and 1N realtions.

```
relation FK2Relation1N {
 i : String;
 idI : Integer;
 lk1 : kaos::Link; lk2 : kaos::Link;
 enforce domain dbmain fk :
  dbmain::ForeignKey {id = idI };
 enforce domain objectiver rel :
  kaos::Relationship { id = i };
 primitive domain idModel : String;
 when {
  idI = toIdInteger(i);
  i = toIdString(idModel, fk.id);
 }
 where {
  lk1.multiplicity = '1';
  lk2.multiplicity = 'n';
  ColumnsFK2Links(cl,lk2, lk1);
 }
}
```

8.5 Applying the bridge

We have taken an excerpt of the automated train control system example presented in [133] to illustrate the bridge between Objectiver and DB-Main. This example shows how KAOS can be used to model the requirements involved in a train traffic security system. The goal model is shown in Figure 8.6 and defines an expectation (`Safe transport`), a main goal to accomplish (`Avoid train collisions`) and several refinements by using simpler requirements. The agents involved in the goals are: `Speed control system`, `On board train controller` and `Tracking system`. The entities managed by the goals are `Train`, `Line` and `TrackSegment`.

The object model is shown in Figure 8.7. As can be observed in this figure, the notation for these models complies with those used in UML for class diagrams. This model represents the entities (`Train`, `Line`, and `TrackSegment`) and the relationships between entities, but agents are not included. `Train` has its own

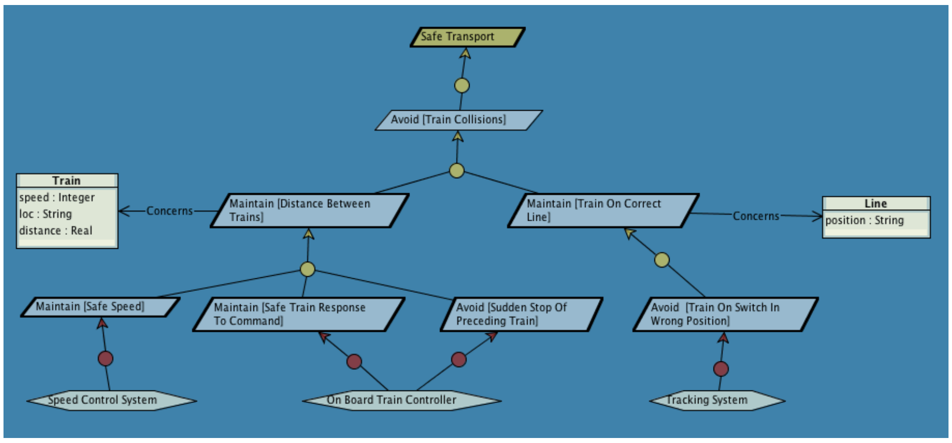

Safe Transport
Avoid [Train Collisions]
Train
speed : Integer
loc : String
distance : Real
Concerns
Maintain [Distance Between Trains]
Maintain [Train On Correct Line]
Concerns
Line
position : String
Maintain [Safe Speed]
Maintain [Safe Train Response To Command]
Avoid [Sudden Stop Of Preceding Train]
Avoid [Train On Switch In Wrong Position]
Speed Control System
On Board Train Controller
Tracking System

Figure 8.6: Goal model of the running example.

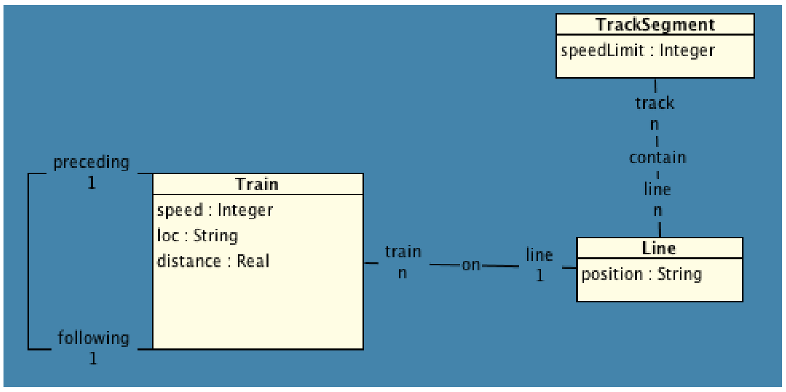


Figure 8.7: Object model of the running example.

association to be able to define the precedence between trains. `Line` includes the `position` attribute which refers to the position inside the track segment (considering that each line has a position in each track). Each `Line` contains one or more `TrackSegments` and each `TracSegment` could be assigned to one or more `Lines` (`contain` association).

To apply our bridge to the previous example, we must firstly export the Objectiver project to an Ecore model, conforming to the Objectiver metamodel (see Figure 8.8). We should then apply the previously presented QVT transformation to obtain the DB-Main model (see Figure 8.9). Next, we have performed the API2MoL extractor to generate the DB-Main database schema, which is shown

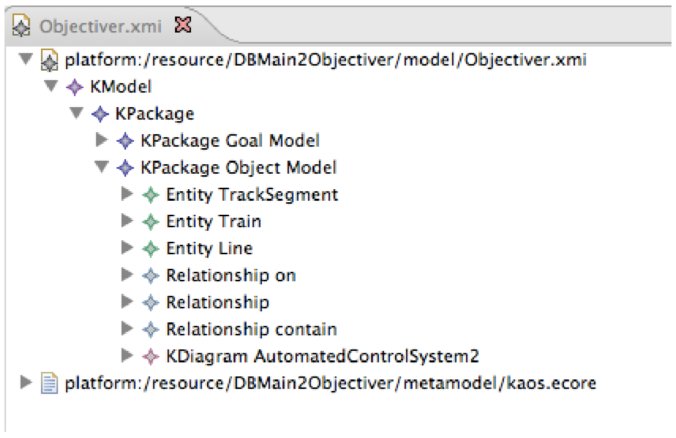

```
Objectiver.xmi
▼ platform:/resource/DBMain2Objectiver/model/Objectiver.xmi
  ▼ KModel
    ▼ KPackage
      ▶ KPackage Goal Model
      ▼ KPackage Object Model
        ▶ Entity TrackSegment
        ▶ Entity Train
        ▶ Entity Line
        ▶ Relationship on
        ▶ Relationship
        ▶ Relationship contain
        ▶ KDiagram AutomatedControlSystem2
▶ platform:/resource/DBMain2Objectiver/metamodel/kaos.ecore
```

Figure 8.8: Ecore model of the Objectiver project.

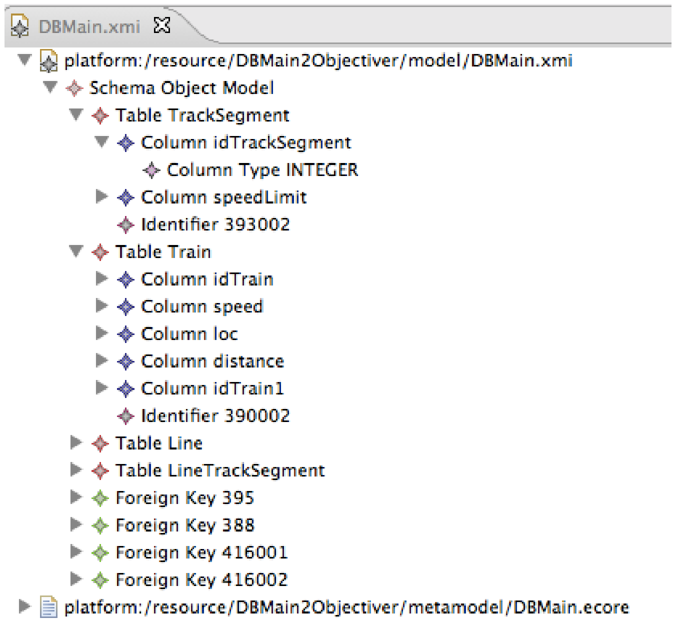

```
DBMain.xmi
▼ platform:/resource/DBMain2Objectiver/model/DBMain.xmi
  ▼ Schema Object Model
    ▼ Table TrackSegment
      ▼ Column idTrackSegment
          Column Type INTEGER
      ▶ Column speedLimit
        Identifier 393002
    ▼ Table Train
      ▶ Column idTrain
      ▶ Column speed
      ▶ Column loc
      ▶ Column distance
      ▶ Column idTrain1
        Identifier 390002
    ▶ Table Line
    ▶ Table LineTrackSegment
    ▶ Foreign Key 395
    ▶ Foreign Key 388
    ▶ Foreign Key 416001
    ▶ Foreign Key 416002
▶ platform:/resource/DBMain2Objectiver/metamodel/DBMain.ecore
```

Figure 8.9: Ecore model of DB-Main after the semantic mapping.

in Figure 8.10. Each relationship has been converted into a foreign key. For the `N:N` relationship a new table has been created (`LineTrackSegment`) along with its foreign keys.

Next, we applied some changes to the generated schema and the object model in order to test the synchronisation. For instance, we added a new number attribute

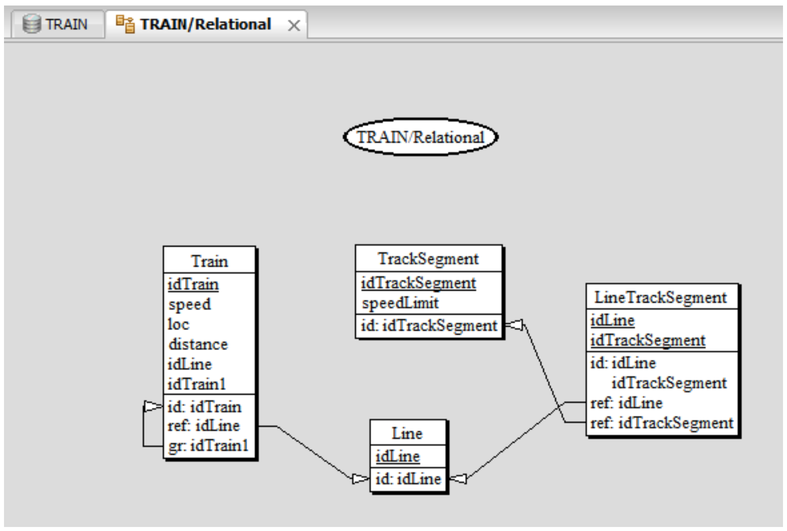
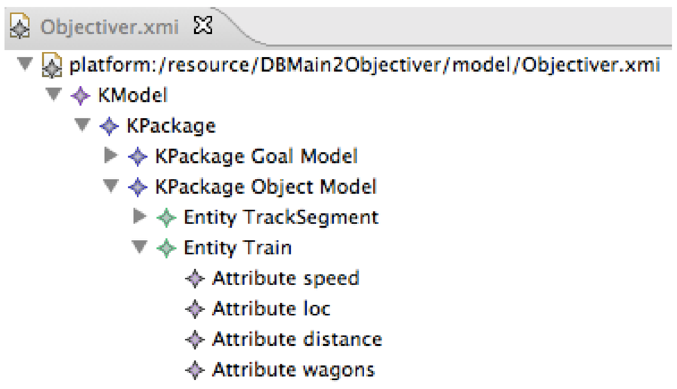


Figure 8.10: DB-Main project extracted by the syntactic mapping.



```
Objectiver.xmi
▼ platform:/resource/DBMain2Objectiver/model/Objectiver.xmi
  ▼ KModel
    ▼ KPackage
      ▶ KPackage Goal Model
      ▼ KPackage Object Model
        ▶ Entity TrackSegment
        ▼ Entity Train
            Attribute speed
            Attribute loc
            Attribute distance
            Attribute wagons
```

Figure 8.11: Ecore model of the Objectiver project after the modification.

to the `Train` table in the DB-Main schema (`wagons`). After applying the injection process and the semantic mapping, the new object model shown in Figure 8.11 was generated.

8.6 Conclusions

We have presented an MDI approach for a case study based on the integration of Objectiver and DB-Main. This integration have allowed us to experiment with the majority of concerns involved in the construction of an MDI bridge: (i) tools may export exchanged data to models or not, (ii) tools can offer several forms to allow access to its data; and (iii) the integration can be unidirectional or bidirectional. In our case, Objectiver provides support to export/import its models to Ecore models, but DB-Main does not offer such support; DB-Main provides three interoperability forms (API, XML and grammar format); and the integration is bidirectional. Therefore, we have explored several implementation strategies for creating injectors and extractors for DB-Main data, as well as the use of QVT Relational to write bidirectional transformations. Some of the main lessons learned in building the Objectiver/DB-Main are the following:

- Automation provided by MDE tools can significantly reduce the implementation effort compared to using GPLs (e.g. Java) and a metamodelling API (e.g. EMF) to build the bridge from scratch. The choice of MDE tool depends on the available data formats, and the criteria introduced in Section 8.3 for injectors and extractors.

- API2MoL and DSL workbenches provide a high level of automation, since they can automatically generate injectors and extractors. However, API2MoL is discontinued and lacks adequate documentation.

- When XML is used, it is worth remarking that the strategies using SAX/-DOM or JAXB can be bidirectional although completely manual, whereas the use of the EMF parser/serialiser only requires writing one or two M2M transformations (depending on the bidirectional supporting) only in case of the metamodel generated by EMF was not valid as pivot metamodel.

- DSL workbenches and EMF's XML injector/extractor are mature tools which are easy to use and learn. Injectors and extractors generated by DSL workbenches that requires the pivot metamodel as input are directly usable, while an additional M2M transformation is normally required for EMF.

- Whenever a grammar format is used to export/import exchanged data, DSL workbench would be the preferable solution if the grammar is not too complicated. In our case, these tools could not be used because the LUN grammar is large and complex.

- Considering the use of QVT, the implementation of a bidirectional mapping is more complicated than a unidirectional one. Owing to the fact that mapping declarations must be applied in two ways, some restrictions have to be considered (e.g. the right side of an assignment can not contain complex runtime expression because it is also the left side of the assignment when the transformation is applied in the opposite way). As writing imperative code is commonly needed, the possibility of defining helper functions in Medini QVT is very useful. The tool allow the combination of imperative code and OCL-style declarative code. Medini QVT provides a lot of useful functionality but the debug support should be improved.

"The road is hard and you are soon tired. As you struggle up a rocky mountain path, a man and a woman ride past on a horse, deep in conversation. As they gallop on, the dust makes you splutter, but they are out of sight before you can react. Eventually you reach the top of the hill, and, for the first time in your life, can look down into the valley beyond. It is fertile and inviting."

from Bloodfeud of Altheus, John Butterfield, David Honigmann and Philip Parker

(Suggested by Juan Manuel)

9 Conclusions

The evolution of legacy systems is a problem that companies must currently address in order to maintain the value of their information systems. Modernisation processes should be applied not only to legacy application migration but also to quality software improvement. Data-intensive information systems are composed of applications, which basically implement the business logic and the GUI, and data which are defined by database schemas. With regard to the data layer of legacy data-intensive systems, wrong design choices or technological limitations may involve quality improvements during the modernisation process. For instance, it would not be possible to implement foreign keys owing to the limitations of the database engine, as in our case study OSCAR. Databases may also need of an appropriate normalisation level in order to ensure the absence of data redundancies and inconsistencies. The objective of this thesis has been to provide a model-driven data reengineering approach that addresses the improvement of the schema quality in the context of the evolution of a legacy information system. As part of the objectives of the thesis we have provided a migration tool in order to define and enact migration processes along with a model-driven interoperability solution with which to integrate data engineering tools into a MDE reengineering approach.

This last chapter will present the conclusions and contributions of the thesis. We shall also outline the future research lines and we shall finalise by presenting the main publications derived from the works contained in this thesis along with other publications, projects and research stays. Below we shall begin to discuss the results attained for each goal and present the fulfillment of the requirements pointed out in Chapter 4.

9.1 Discussion

In Chapter 1 we introduced the four main goals of this thesis: (G1) the implementation of a data reengineering process by using MDE techniques; (G2) employing different strategies in order to elicit foreign keys for the restructuring stage of the process; (G3) building a tool that is able to automate the development of model-driven reengineering processes and (G4) tackling the MDE-base tool interoperability through the building of a bidirectional bridge. The solution architecture we devised for each goal was presented in Chapter 4, and we shall now start by first tackling the general requirements identified in Section 4.1. We shall then discuss to what extent the specific requirements of each goal are accomplished. A table will then be used in order to summarise how all the requirements have been fulfilled in which the first column identifies the requirement; the second column points out the solution that achieves the fulfillment; the third column indicates the degree of fulfillment (low, medium and high); and the last column includes other considerations about the fulfillment.

We shall first remind the reader of the general requirements defined for the approach presented as a result of this thesis, which are described in Section 4.1: (R1) *Productivity*, (R2) *Automation*, (R3) *Modularity, Extensibility and Reusability*, (R4) *Evolvability*, (R5) *Consistency* and (R6) *Technology Independence.*

We have profited from the benefits provided by MDE in order to meet the *R1*, *R2*, *R3*, *R4*, *R5* and *R6* requirements. In general, metamodelling and model transformations are useful techniques that provide several benefits during the implementation of a solution.

Productivity (requirement *R1*) is one of the main reasons for applying model-driven techniques rather than a traditional solution. As we noted in the assessment of our data reengineering approach (Chapter 6), productivity is the main factor that encourages the use of MDE [5]. The savings in effort made when applying MDE during the building of a migration process tool were similarly quantified in Chapter 7 and compared to a traditional solution. Metamodels and model transformations enable the building of generative architectures with which to represent, transform and automatically generate data and software artefacts. Migration tools that support the definition and enactment of reengineering process is another of the contributions included in our proposal that allows productivity to be improved.

Model transformation allows the developer to automate tedious and repetitive tasks (requirement *R2*), that usually have to be applied several times in a reengineering process. For example, in a migration scenario, the same reengineering process has to be applied to a large number of legacy applications, which means that a traditional manual approach results in high-cost software solutions.

Metamodels and model transformation chains enable modularity of solutions and can be easily reused (requirement *R3*) in another different solution to that for which they were created. In order to reuse a chain, the inputs and outputs of this transformation chain must be connected to another solution by injecting/-transforming them in accordance with the proper formats required by the other solution. A modular and reusable architecture facilitates extensibility because a new functionality can be developed and added to a model-driven solution by taking into account the metamodels that act as inputs and outputs of modules or process stages.

Using MDE in order to build software could improve the capabilities of the integration between the software and database subsystems, i.e. the synchronisation between programs and data could be facilitated through coupled model transformations. A transformation chain could therefore allow the consistency between database and data access code to be maintained: changes in data should trigger changes in data access code and vice versa. Database evolution could therefore guarantee the right evolution of programs (requirement *R4*). Model techniques used in our solution promotes evolvability by means of traces models along with model transformations with which to propagate database changes. Our proposal has not yet implemented either the generation of trace models or the corresponding model transformations with which to map the tracing data. However, the data reengineering solution proposed has considered the consistency (requirement *R5*) between data and the ORM definitions because changes resulted from the restructuring stage are propagated to the data definitions (DDL scripts) and ORM definitions (JPA access code).

Finally, some previous fulfillments in requirements provide technology independence (requirement *R6*). Our process was designed to take a database representation at a high level (Data model and Defect model) and regenerate a new one after providing data quality, without considering source or target database technologies. Model injectors and artefact generators have to deal with problems concerning specific or their own formats and technologies.

Table 9.1 shows a summary of the general requirements of the approach presented and the considerations to bear in mind.

Req.	**Solution**	**Fulfillment**	**Other considerations**
R1	Metamodels, model transformations, tooling	High	None
R2	Model transformation chain	High	None
R3	Metamodels and model transformation chain	High	None
R4	Trace models and model transformation chain	None	Not implemented yet
R5	Model Transformations	Medium	Only for JPA definitions
R6	High level models	High	Inject./gener. technology-specific

Table 9.1: Fulfilment of the general requirements

9.1.1 G1. Data Reengineering Process

The requirements defined for the goal *G1*, that is the data reengineering process, are the following: (R7) *Representing a database at a high abstraction level*, (R8) *Providing the migration stakehorlders with manual support in order to manage the schema conversion* and (R9) *Variability in the artefacts generated.*

The models have been useful to explicitly represent the database information that is discovered in the reverse engineering stage (requirement *R7*), which is described at a high abstraction level by means of metamodels that we have created as, for instance, a *Defect model* or a *Functional Dependency model.* Several implicit constructors or semantic concepts are discovered and modelled from database records and schema; these models are used in the restructuring stage during which the data quality improvements are applied. We have taken advantage of models and metamodels as formalisms that provide a uniform representation of any kind of information.

The database administrators' and developers' knowledge must be taken into account when deciding what implicit FKs should be defined (requirement R8). As discussed at the end of Chapter 5, the results achieved by each separate analysis are not conclusive. Even the manual combination of the results does not ensure that every implicit constraint is well proposed or all implicit constraints have been discovered. In order to improve the reliability of the foreign key elicitation process, we have therefore provided a wizard to assist the migration developers in the task that consisting of confirming or rejecting the implicit foreign key discovered after applying our approach. It is worth noting that some implicit FKs could not be discovered by our approach

Our data reengineering process has considered the generation of several kinds of artefacts (requirement *R9*), such as scripts with which to regenerate the database and data access code for an ORM technology (e.g. JPA) and data access code implemented on a ORM technology (JPA). Moreover, other generators can easily be integrated into our solution, as noted in Section 6.7.5. Table 9.5 provides a summary of the discussion concerning the requirements of the G1 goal by using the three criteria established.

Req.	**Solution**	**Fulfillment**	**Other considerations**
R7	Data and Defect models	Medium	None
R8	Wizard to confirm FKs	Medium	Only for FKs proposed
R9	Different generators provided	High	Extensible architecture

Table 9.2: Fulfilment of the requirements of goal G1

9.1.2 G2. Strategies of FK Discovering

We shall now remind the reader of the requirements defined for the second high goal *G2*, that is the strategies of FK which discover: (R10) *Use of multiple sources for the analysis*, (R11) *Use of different techniques for the analysis* and (R12) *Combining results from the different analysis.*

The elicitation of foreign keys is achieved by harvesting knowledge from different sources (requirement *R10*), such as data stored in databases, schemas that define the structure of data and database access code in applications. Several sources allowed us to apply different strategies. On some occasions, these provided simultaneous results which proved to be trustworthy, while on others they provided complementary results with which to elicit new constraints. However, some alternative sources have not been addressed, such as reports or screen layouts.

The data and schema analysis have been implemented (requirement *R11*) by using two strategies: (i) injecting data into models and applying model transformations, and (ii) directly accessing the database by performing algorithms written in PL-SQL. We experimented with both implementations and this allowed us to provide an assessment of each one.

As noted in Chapter 5, we have manually combined the results attained by using each kind of analysis (requirement *R12*). In that chapter we discussed the inappropriateness of only analysing one source of the legacy application, owing to the fact that the different sources of information have different levels of reliability and there is no perfect source of information that would be sufficient in the foreign key elicitation. Therefore, the triangulation of the results obtained from the different analyses would therefore appear to be promising. Table 9.3 shows a summary of the fulfillments of the previous requirements.

Requirement	**Solution**	**Fulfillment**	**Other considerations**
R10	Data, schema and code	High	Reports or Screens not considered
R11	DAS-M and DAS-D	High	Assessment provided
R12	Triangulation of results	High	Manual step

Table 9.3: Fulfilment of the requirements of goal G2

9.1.3 G3. Migration Tool

We shall first remind the reader of the requirements defined for the goal *G3*, that is the migration tool: (R13) *Supporting the process definition*, (R14) *Ability to define migration processes in abstract and concrete form*, (R15) *Supporting the process enactment*, (R16) *Integration with existing development environments*, (R17) *Tasks should define the dependencies among them*, and (R18) *Support for task assignment.*

It is worth noting that legacy applications are usually composed of thousands of artefacts that should be migrated to the new platform. A migration tool should automate how a reengineering process is applied to each of the software artefacts in the legacy system. The tool presented in this thesis is capable of taking the legacy resources and applying the right process for each one, according to the type of legacy resource and the subprocess in the migration process which is in charge of transforming this kind of artefact.

As part of this thesis, we have developed a tool in order to support the definition and enactment of model-driven software processes. We have, more specifically, focused on the domain of migration processes, characterised by the need to be applied to a large amount of artefacts comprising the legacy application. This scenario requires a specific solution that supports the definition (requirement *R13*) and enactment (requirement *R14*) of these processes. Our tool offers a SPEM-like software process language whose objective is to define model-driven migration processes, but could be applicable to any kind of process. The migration processes defined with this language (i.e. abstract migration models) are instantiated in the concrete migration models which include all the information needed (e.g. tools and database scripts) to be enacted .

We have created an process interpreter that takes as input the concrete models and enacts the process (requirement *R15*) by providing automated and manual tasks on a Trac system as tickets. The automated tasks are executed from Trac by using ANT scripts and the manual tasks are supported by Mylyn tasks which can be integrated into a development environment as Eclipse, thus providing the task with contextual information to the task (requirement *R16*). The Trac system is also able to establish dependencies among tasks (requirement *R17*), which means that a task can be enacted (i.e. executed by ANT or unblocked in Mylyn) only if all the preceding tasks in its dependency have already been completed. Finally, the task assignment (requirement *R18*) is provided by Mylyn and its Eclipse plugin.

The accomplishment of the requirements is summarised in Table 9.4.

Requirement	**Solution**	**Fulfillment**	**Other considerations**
R13	SPEM-like process language	High	None
R14	Abstract and concrete processes	High	None
R15	ANT scripts and Mylyn tasks	High	None
R16	Mylyn plugin in Eclipse	High	None
R17	Trac ticket dependencies	High	None
R18	Mylyn plugin in Eclipse	High	None

Table 9.4: Fulfilment of the requirements of goal G3

9.1.4 G4. Tool Interoperability

Finally, we shall remind the reader of the requirements defined for goal *G4*, the tool Interoperability: (R19) *Bidirectionality*, and (R20) *Dealing with different scenarios*.

We first experimented with a model-driven bridge with the objective of integrating the ConExp tool into our reengineering process with the purpose of obtaining functional dependencies. We then built a bridge in order to integrate the DB-Main tool into MDE solutions (requirement *R20*). This bridge was tested by the case study. Finally we tackled how DB-Main could interoperate with other tools, in particular Objectiver, by means of a bidirectional bridge (requirement *R19*). The building of these bridges enabled us to experiment with different strategies so as to implement the syntax mapping (i.e. injectors and extractors). We were additionally able to investigate the capabilities of QVT Relational as regards creating bidirectional semantic mappings in the case of the DB-Main/Objective bridge.

Requirement	**Solution**	**Fulfillment**	**Other considerations**
R19	Injectors/Extractors and QVT transformation	High	None
R20	DB-Main/modelsware and Objectiver/DB-Main	High	None

Table 9.5: Fulfilment of the requirements of goal G4

9.2 Contributions

The contributions of this thesis are described below. They will be categorised according to the goals identified in Section 1.2.

9.2.1 G1. Data Reengineering Process

As noted in [18], one of the challenges of database engineering (e.g. data reengineering) is ”how to integrate transformational database engineering into emerging MDE frameworks”. This work contributes to this endeavour by evaluating the extent to which MDE techniques can confront data reengineering challenges. To the best of our knowledge, this work is one of the first contributions to provide an assessment of the use of MDE in data reengineering. We have addressed the problem of discovering and removing defects in database schemas in a database migration scenario. In particular, we have considered two defects as proof of concept: undeclared foreign keys and disabled constraints. We have defined an MDE-based approach for this setting which has been implemented as a reengineering process in which the three aforementioned stages are realised as model transformation chains. This approach is showcased by means of an information system that is widely used in the healthcare industry in Canada: *OSCAR* (Open Source Clinical Application Resource) [111]. We have contrasted our work with the tasks usually performed in traditional approaches and have identified some benefits and drawbacks of applying MDE techniques in data reengineering, which could enable us to assess to what extent they could be applicable to other problems in this area. The contribution is therefore twofold: the model-driven reengineering approach described and the assessment presented.

9.2.2 G2. Strategies of FK Discovering

The data reengineering process (schema conversion) provided in this work has been showcased by the OSCAR system. In contrast to many other research works that start by proposing a new or improved solution to the data reengineering problem, followed by a validation with problem case studies (often handpicked to make a point), we start by studying the actual problem in the context of a real-world, large-scale legacy system in the healthcare industry. As a result of our analysis, we find that many of the assumptions commonly made in DB reengineering methods and tools do not readily apply in practice. Based on our problem analysis we devise a process with which to reengineer legacy information systems as regards establishing referential integrity constraints and combining existing reengineering methods. In summary, our results suggest that the process of reengineering legacy information systems with regard to establishing referential integrity constraints may be considerably more complex than is commonly assumed. It must be understood as an *incremental* detection process.

9.2.3 G3. Migration Tool

Owing to the lack of software environments for model-driven reengineering processes, we have built a tool, named *Models4Migration* in order to partly automate the migration effort. This tool is based on an MDE approach that is implemented around three main design choices: i) models are used to define migration processes between pairs of specific technologies, ii) these process models are enacted through a model transformation chain that generates the automated and manual tasks to be performed, and iii) the automated tasks are directly executed and the manual tasks are generated as Trac [1] tickets which are integrated as tasks into the Mylyn tool [2]. In [134] is stated that *"It is desirable to define software processes with sufficient precision so that many of the routine enactment tasks can be automated"*, and the tool proposed has explored how MDE techniques can be useful to automate enactment tasks in software development processes, particularly in migration processes. For instance, creating tickets is a tedious and time consuming task performed by team leaders, so the automated generation of tickets is one of the benefits of our tool. We have chosen Trac and Mylyn as they are open-source tools commonly used by software companies, but tools with a similar functionality could be used in our approach. The model-driven strategy applied allows us to obtain a technology-independent approach, as the tool is configured by means of models that represent both the migration processes and the behaviour to be performed in order to enact them. These models are encapsulated whitin cartridges that are plugged into the tool.

The tool proposed therefore makes three contributions. Firstly, to the best our knowledge, Models4Migration is the first proposal to enact process models by executing automated tasks (e.g. model transformations) and generate programming manual tasks which are integrated into a task management tool (i.e. Mylyn). Note that our approach goes beyond the definition of software processes provided by

[1]http://trac.edgewall.org

[2]http://www.eclipse.org/mylyn

tools such as EPF [3], the management of processes realised with Microsoft Project [4] or the enactment implemented for some existing process modelling languages [91]. However, a migration process also involves financial and technical constraints, resource planning and risk management among other activities, which are not whitin of the scope of this work. Secondly, our work is one of the first experiences to show how an MDE approach can be used to build a tool supporting software development processes from the definition of software processes to the management of the tasks to be performed by managers and developers. Thirdly, we present a solution to support migration processes that have been implemented with MDE technologies.

9.2.4 G4. Tool Interoperability

In order to ease the integration of external tools, such as DB-Main or Concept Explorer, into the data reengineering process we have devised a model-based architecture that aims at bridging the gap between tools. Each tool or process usually provides its data by using different formats and software artefacts (APIs, files or other specific resources). For instance, *DB-Main* offers only a data accessing API along with two different files in two different formats (one XML file and one project file). The MDE techniques have proved to be useful as regards easing and extending the interoperability capabilities of DB-Main. By implementing the adequate artefacts of which the architecture of a bridge is composed, data could be exchanged from/to DB-Main. We take advantage of the existence of three different means to access DB-Main in order to implement several alternatives so as to extract or inject data from DB-Main. Through this case study, we therefore contribute to analysing and discussing how MDE can address several interoperability scenarios.

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

[4]https://products.office.com/en-us/project (accessed on 11-20-2014)

9.3 Future work

We have arranged the future work into several categories according to the goals defined in Chapter 1.

9.3.1 G1. Data Reengineering Process

Future work could focus on two directions: improving the reengineering process applied and detecting more defects. On the one hand, some issues to be considered are: (i) tackling the data and program conversions in the approach proposed by creating the coupled transformations needed in order to synchronise the evolution of data and code; (ii) this data conversion could involve the integration of ETL tools into an MDE solution; (iii) the current foreign key detection could be improved by applying a dynamic code analysis which would avoid the limitation of having the SELECT fragments ordered, in addition to providing a more trustworthy strategy; (iv) support for constraints implemented as triggers, signifying that old data could be kept in the new database (rather than being excluded by using a residual schema) although they do not satisfy new constraints. With regard to the program conversion, it is worth noting that new problems would arise when adding the constraints detected. For instance, it would be necessary to check a proper order in which to delete those sentences that affect the tables involved in a new foreign key. On the other hand, the experience gained in our work could be used to tackle the development of a framework with which to improve the quality of database schemas. For this, we would have to bear in mind the conceptual framework proposed in [78] and the defect taxonomy presented in [79]. Finally, NoSQL systems have emerged as an alternative to relational systems for the management of huge collections of complex data in modern applications that require high scalability [135]. New data reengineering scenarios, such as migration from relational systems to NoSQL systems or NoSQL reverse engineering, are therefore appearing. MDE can be applied in these scenarios as is shown in [136], in which a model-driven approach is used to infer the schema of NoSQL databases.

9.3.2 G2. Strategies of FK Discovering

We anticipate several directions for future work in the context of RIC reengineering. First, we intend to further investigate the OSCAR case study, and to involve the developers in the establishment of an, albeit partial, ground truth. Second, we plan to consider other sources of information for the identification and ranking of RIC candidates. We are particularly thinking of integrating historical information. For instance, let us assume that the historic analysis reveals that the same developer has created both tables involved in a RIC candidate, this could be seen as an additional confirmation argument. In contrast, if a RIC candidate involves two very recently created tables, the names of which do not appear in the Hibernate file nor in the JPA annotations, this could be considered as a rejection argument. Last but not least, we intend to devise a tool-supported methodology with which to assist developers to incrementally implement RIC candidates identified in a legacy software system.

9.3.3 G3. Migration Tool

Future work includes, among other things:

- The extension of our architecture in order to support any kind of software process rather than only model-driven migration processes. At this moment, our process definition language is aimed at model-driven tasks but the *Migration model* could be extended with new `MigrationTaskUse`, `MigrationToolUse` and `MigrationWorkProductUse`. It would also be necessary to extend the enactment by adding new task generators for ANT and/or Mylyn.
- The support for complex dependencies between tasks. For example, the task dependency mechanism now implemented considers only whether a task is blocked (completed) or unblocked (uncompleted). However, we do not check whether all the expected artefacts were generated after the execution of a task.
- The integration of external tools (e.g. word processors) for the realisation of manual tasks.

- Implementing a traceability mechanism to keep track of the source elements (defined in the Abstract Migration model) in the Task model used to generate the automated and manual tasks. When migrated data are not the expected data or they have failures, the traces of the transformation chain could be crucial in order to identify failures in the tasks executed or simply bad decisions made during the definition of the migration process.

9.3.4 G4. Tool Interoperability

Future work could focus on two main directions: creating new injectors and extractors along with creating bidirectional semantic mappings. At this moment, a review on existing injection and extraction techniques would be very useful, as would an evaluation of the existing tools for the most widely used formats. With regard to bidirectional mappings, the capabilities of QVT Relational should be investigated thoroughly.

On the other hand, the existence of guidelines on building MDI bridges could help to developers address this issue. Moreover, more case studies are needed in order to illustrate the advantages and drawbacks of using MDE to integrate tools.

9.4 Publications related to the thesis

9.4.1 Journals with an impact factor

- Francisco Javier Bermúdez Ruiz, Jesús García Molina, Óscar Díaz García *A model-driven reengineering approach for the schema conversion.* Journal of Information Systems (*under review*).

- Francisco Javier Bermúdez Ruiz, Óscar Sánchez Ramón, Jesús García Molina *A model-driven tool to support the definition and enactment of migration processes.* Journal of Software and Systems (*under review*).

9.4.2 International conferences and workshops

- Loup Meurice, Francisco Javier Bermúdez Ruiz, Jens H. Weber, Anthony Cleve, *Establishing referential integrity in legacy information systems - Re-*

ality bites!. In the proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'14), Victoria (Canada), 2014.

- Francisco Javier Bermúdez Ruiz, Jesús García Molina, Óscar Díaz García, *Data Integration between Objectiver and DB-Main: A case study of a model-driven interoperability bridge. Submitted* to the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'16), Rome (Italy), 2016 (*author notification is November, 26th*)

- Francisco Javier Bermúdez Ruiz, Óscar Sánchez Ramón, Jesús García Molina, *Definition of processes for MDE-based migrations.* In the proceedings of the 3rd International Workshop on Process-Based Approaches for Model-Driven Engineering (PMDE'13), Montepellier (France), 2013.

- Jens H. Weber, Anthony Cleve, Loup Meurice, Francisco Javier Bermúdez Ruiz, *Managing Technical Debt in Database Schemas of Critical Software.* In the proceedings of the 4th International Workshop on Managing Technical Debt (MTD'14), Victoria (Canada), 2014.

9.4.3 National conferences

- Francisco Javier Bermúdez Ruiz, Jesús García Molina, *Un framework para la modernización de datos relacionales dirigida por modelos.* In proceedings of the XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'12), Almería (Spain), 2012.

- Óscar Sánchez Ramón, Francisco Javier Bermúdez Ruiz, Jesús García Molina, *Experiencias de Modernización de Software con DSDM.* In Proceedings of the XVIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'13), Madrid (Spain), 2013.

- Francisco Javier Bermúdez Ruiz, Jesús García Molina, Óscar Díaz García *DB-Main/Models: Un caso de estudio sobre la interoperabilidad de herramientas basada en MDE.* In Proceedings of the XIX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'14), Cádiz (Spain), 2012.

9.5 Projects that are related to this thesis

- **“MOMO: Un Entorno de Modernización de Software Dirigida por Modelos en Escenarios de Migración de Plataformas (Ref. 08797/PI/08)”**. Granted by the Fundación Séneca (Regional plan of Science and Technology 2007-2010). From 2009-01-01 until 2010-12-31. In this project we designed the first approach for inferring the layout of the Oracle Forms windows.

- **“Impulso de la Investigación en Tecnologías del Desarrollo de Software (Un Entorno para el Desarrollo y Modernización Basado en Modelos: Forms-ADF) (Ref. CARM 129/2009)”**. Granted by the Consejería de Universidades, Empresas e Investigación. From 2009-06-04 until 2010-12-31. The goal of this project was the definition of a software environment for the migration of Oracle Forms applications to ADF. We used the results obtained in the previous project in order to implement the layout inference engine.

- **“GUIZMO: Un framework para la modernización basada en modelos de interfaces de usuario”**. Granted by the Fundación Séneca (Research Projects Funds). From 2011-01-01 until 2014-12-31. In this project we tackled the development of a model-driven framework for analysing the code of event handlers in order to separate the concerns that are tangled. Moreover, during this project we created a tooling to assist the automatic generation of web interfaces from wireframes.

9.6 Research stays

- **Research Stay in the Université of Namur (Belgium)**, during 3 months, in the Precise research group. We were working in applying our approach to a real case study as the OSCAR system and we also collaborated in a work about combining several techniques for the foreign key elicitation that resulted in [108]. We also were working in integrating the DB-Main data engineering tool in our approach by defining an MDI bridge.

9.7 Transfer of technology

- **“Herramienta orientada a la migración basada en modelos”**. Granted by the Ministerio de Industria, Turismo y Comercio. CDTI project granted to the Sinergia IT (Deusto Group) software company. From 2010-01-01 until 2011-12-31. This project was aimed at the creation of a tooling to assist the automatic migration of Oracle Forms applications to a Java platform. Our research group collaborated with the Sinergia IT company to accomplish research tasks in the context of this project.

References

[1] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Developing legacy system migration methods and tools for technology transfer. *Softw. Pract. Exper.*, 38(13):1333–1364, November 2008.

[2] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] Kathi Hogshead Davis and Peter H. Aiken. Data reverse engineering: A historical survey. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'2000)*, page 70, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0881-2.

[4] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *IEEE Computer*, 43(8):110–112, 2010.

[5] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[6] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[7] OMG. *MDA Guide Version 1.0.1. Object Management Group (OMG).* http://www.omg.org/mda, 2003.

[8] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling.* Wiley, 2008.

[9] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Autom. Softw. Eng.*, 21(2):147–186, 2014.

[10] Jose-Norberto Mazón and Juan Trujillo. A model driven modernization approach for automatically deriving multidimensional models in data warehouses. volume 4801 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2007.

[11] Javier Cánovas and Jesús Molina. An architecture-driven modernization tool for calculating metrics. *IEEE Softw.*, 27(4):37–43, July 2010. ISSN 0740-7459.

[12] F. Fleurey et al. Model-driven engineering for software migration in a large industrial context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the MoDELS'07*, volume 4735 of *LNCS*, pages 482–497. Springer, 2007.

[13] Thijs Reus, Hans Geers, and Arie van Deursen. Harvesting software systems for mda-based reengineering. In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'06, pages 213–225, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. ISBN 0123749131, 9780123749130.

[15] Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Mario Piattini, and Christof Ebert. Reengineering technologies. *IEEE Software*, 28(6): 13–17, 2011.

[16] OMG. Architecture-Driven Modernization. http://adm.omg.org/.

[17] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wad, D. O'Sullivan, and R. Richardson. Legacy System Migration: A Legacy Data Migration Engine. In Ed. Czechoslovak Computer Experts, editor, *Proceedings of the 17th International Database Conference (DATASEM'1997)*, pages 129–138, 1997.

[18] Jean-Luc Hainaut. Transformation-based database engineering. In Laura C. Rivero, Jorge Horacio Doorn, and Viviana E. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 707–713. Idea Group, 2005.

[19] J. Bezivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2006.

[20] Jean Bézivin, Hugo Brunelière, Jordi Cabot, Guillaume Doux, Frédéric Jouault, and Jean-Sébastien Sottet. Model Driven Tool Interoperability in Practice. In *Proceedings of the 3rd Workshop on Model-Driven Tool & Process Integration (co-located with ECMFA 2010)*, pages 62–72, June 2010.

[21] M. R. Blaha and W. J. Premerlani. Observed idiosyncracies of relational database designs. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'1995)*, page 116, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7111-4.

[22] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pages 158–165. IEEE Computer Society Press, 1991. ISBN 0-89791-391-4.

[23] L. Osterweil. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 2–13. IEEE Computer Society Press, 1987. ISBN 0-89791-216-0.

[24] Volker Gruhn. Process-centered software engineering environments, a brief history and future challenges. *Ann. Software Eng.*, 14(1-4):363–382, 2002.

[25] OMG. Software & Systems Process Engineering Metamodel Specification (SPEM). Technical report, "OMG", April 2008. URL http://www.omg.org/spec/SPEM/2.0.

[26] Reda Bendraou, Marie-Pierre Gervais, and Xavier Blanc. Uml4spm: A uml2.0 based metamodel for software process modelling. In *Model Driven Engineering Languages and Systems*, volume 3713, pages 17–38. Springer Berlin Heidelberg, 2005.

[27] Reda Bendraou, Benoît Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an executable spem 2.0. In *APSEC*, pages 390–397, 2007.

[28] Ralf Ellner, Samir Al-Hilank, Johannes Drexler, Martin Jung, Detlef Kips, and Michael Philippsen. espem: a spem extension. In *Modelling Foundations and Applications*, volume 6138, pages 116–131. Springer Berlin Heidelberg, 2010.

[29] Henrik Steudel, Regina Hebig, and Holger Giese. A build server for model-driven engineering. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, pages 67–72. ACM, 2012.

[30] DB-MAIN. The DB-MAIN official website. http://www.db-main.be, 2011.

[31] Serhiy Yevtushenko. System of data analysis concept explorer. In *Proceedings of the 7th National Conference on Artifical Intelligence KII-2000*, pages 127–134. Proceedings of the 7th National Conference on Artifical Intelligence KII-2000, 2000.

[32] Viorica Varga and Katalin Tünde Jánosi Rancz. A software tool to transform relational databases in order to mine functional dependencies in it using formal concept analysis. In *Proceedings of the Sixth International Conference on Concept Lattices and Their Applications*, pages 1–9, 2008.

[33] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991. ISBN 1559370793.

[34] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[35] Jean Henrard, Jean-Marc Hick, Philippe Thiran, and Jean-Luc Hainaut. Strategies for data reengineering. In Arie van Deursen and Elizabeth Burd, editors, *WCRE*, pages 211–220. IEEE Computer Society, 2002. ISBN 0-7695-1799-4.

[36] Sinergia Tecnológica (Oesia group) and Modelum (University of Murcia). Herramienta orientada a la migración basada en modelos. CDTI project, Ministry of Industry, Turism and Comerce. 2010-2011.

[37] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, 24(3):45–77, December 2007.

[38] Vijay K. Vaishnavi and William Kuechler, Jr. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach Publications, Boston, MA, USA, 1st edition, 2007. ISBN 1420059327, 9781420059328.

[39] OMG. Query/View/Transformation 1.1. http://www.omg.org/spec/QVT/, 2011.

[40] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010. ISBN 0136086209, 9780136086208.

[41] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[42] Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, pages 154–. IEEE Computer Society, 1998.

[43] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. Emf: Eclipse modeling framework 2.0, 2009.

[44] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, June 2013. URL `http://www.omg.org/spec/MOF/2.4.1`.

[45] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006. ISSN 0740-7459.

[46] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 173–174. ACM, 2010.

[47] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software and System Modeling*, 13(2):713–734, 2014.

[48] Javier Luis Cánovas Izquierdo, Oscar Diaz, Gorka Puente, and Jesus Garcia Molina. Schemol: Un lenguaje especifico del dominio para extraer modelos de bases de datos relacionales. In *Póster en las XI Jornadas de Ingeniería del Software y Bases de Datos*, 2011.

[49] Markus Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[50] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, 1992.

[51] Hugo Bruneliere et al. Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *EC-MFA*, volume 6138 of *LNCS*, pages 32–47. Springer, 2010. ISBN 978-3-642-13594-1.

[52] Perdita Stevens. A simple game-theoretic approach to checkonly qvt relations. *Software and System Modeling*, 12(1):175–199, 2013.

[53] Dimitris Kolovos, Richard Paige, and Fiona Polack. The epsilon transformation language. In *Proceedings of the First International Conference on Theory and Practice of Model Transformations*, ICMT 2008, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] J.-L. Hainaut, M. Chandelon, C. Tonneau, and M. Joris. Contribution to a theory of database reverse engineering. In *Proceedings of the IEEE Working Conf. on Reverse Engineering*, pages 161–170, Baltimore, May 1993. IEEE Computer Society Press.

[55] Oreste Signore, Mario Loffredo, Mauro Gregori, and Marco Cima. Reconstruction of er schema from database applications: a cognitive approach. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'1994)*, pages 387–402. Springer-Verlag, 1994. ISBN 3-540-58786-1.

[56] Jean-Marc Petit, Farouk Toumani, and Jacques Kouloumdjian. Relational database reverse engineering: A method based on query analysis. *Int. J. Cooperative Inf. Syst.*, 4(2-3):287–316, 1995.

[57] Jens H. Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 193–210. Springer-Verlag New York, Inc., 1997. ISBN 3-540-63531-9.

[58] Hongji Yang and William C. Chu. Acquisition of entity relationship models for maintenance-dealing with data intensive programs in a transformation system. *J. Inf. Sci. Eng.*, 15(2):173–198, 1999.

[59] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovery of "interesting" data dependencies from a workload of sql statements. In *Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery (PKDD'1999)*, pages 430–435. Springer-Verlag, 1999. ISBN 3-540-66490-4.

[60] Jianhua Shao, Xingkun Liu, G. Fu, Suzanne M. Embury, and W. A. Gray. Querying data-intensive programs for data design. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'2001)*, pages 203–218. Springer-Verlag, 2001. ISBN 3-540-42215-3.

[61] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'2006)*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1.

[62] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. SQL2XMI: Reverse engineering of UML-ER diagrams from relational database schemas. In Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta, editors, *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 187–191, 2008.

[63] Anthony Cleve, Jean-Roch Meurisse, and Jean-Luc Hainaut. Database semantics recovery through analysis of dynamic SQL statements. *Journal on Data Semantics*, 15:130–157, 2011.

[64] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. Dynamic program analysis for database reverse engineering. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science. Springer, 2012. to appear.

[65] V. M. Markowitz and J. A. Makowsky. Identifying extended entity-relationship object structures in relational schemas. *IEEE Trans. Softw. Eng.*, 16(8):777–790, 1990. ISSN 0098-5589.

[66] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Commun. ACM*, 37(5):42–ff., 1994. ISSN 0001-0782.

[67] Roger H. L. Chiang, Terence M. Barron, and Veda C. Storey. Reverse engineering of relational databases: extraction of an eer model from a relational database. *Data Knowl. Eng.*, 12(2):107–142, 1994. ISSN 0169-023X.

[68] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Inf. Syst.*, 27(1):1–19, 2002. ISSN 0306-4379.

[69] Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Data Min. Knowl. Discov.*, 16(2):197–219, 2008. ISSN 1384-5810.

[70] Nattapon Pannurat, Nittaya Kerdprasop, and Kittisak Kerdprasop. Database reverse engineering based on association rule mining. *CoRR*, abs/1004.3272, 2010.

[71] Joobin Choobineh, Michael V. Mannino, and Veronica P. Tseng. A form-based approach for database analysis and design. *Communications of the ACM*, Vol. 35, N2:108–120, 1992.

[72] James F. Terwilliger, Lois M. L. Delcambre, and Judith Logan. The user interface is the conceptual model. In *Proceedings of 25th International Conf. on Conceptual Modeling (ER'2006)*, volume 4215 of *Lecture Notes in Computer Science*, pages 424–436. Springer, 2006.

[73] Ravi Ramdoyal, Anthony Cleve, and Jean-Luc Hainaut. Reverse engineering user interfaces for interactive database conceptual analysis. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'2010)*, volume 6051 of *Lecture Notes in Computer Science*. Springer, 2010.

[74] Jean-Marc Petit, Jacques Kouloumdjian, Jean-Francois Boulicaut, and Farouk Toumani. Using queries to improve database reverse engineering. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'1994)*, pages 369–386. Springer-Verlag, 1994. ISBN 3-540-58786-1.

[75] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Ugo de Carlini. Recovering class diagrams from data-intensive legacy systems. In *Proceedings of the 16th IEEE International Conference on Software Maintenance (ICSM'2000)*, page 52. IEEE Computer Society, 2000. ISBN 0-7695-0753-0.

[76] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, University of Namur, 2003.

[77] OMG. *Knowledge Discovery Meta-Model (KDM) v1.0*. http://www.omg.org/spec/KDM/1.0/, 2008.

[78] Jonathan Lemaitre and Jean-Luc Hainaut. Transformation-based framework for the evaluation and improvement of database schemas. In *Proceedings of the 22Nd International Conference on Advanced Information Systems Engineering*, CAiSE'10, pages 317–331. Springer-Verlag, 2010.

[79] Jonathan Lemaitre and Jean-Luc Hainaut. Quality evaluation and improvement framework for database schemas - using defect taxonomies. In Haralambos Mouratidis and Colette Rolland, editors, *CAiSE*, volume 6741 of *Lecture Notes in Computer Science*, pages 536–550. Springer, 2011.

[80] Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Danilo Caivano, and Mario Piattini. Database schema elicitation to modernize relational databases. In *ICEIS (1)*, pages 126–132. SciTePress, 2012. ISBN 978-989-8565-10-5.

[81] Raghavendra Reddy Yeddula, Prasenjit Das, and Sreedhar Reddy. A model-driven approach to enterprise data migration. In *Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, pages 230–243, 2015.

[82] J-L Hainaut, Vincent Englebert, Jean Henrard, J-M Hick, and Didier Roland. Database evolution: the db-main approach. In *Entity-Relationship Approach—ER'94 Business Modelling and Re-Engineering*, pages 112–131. Springer, 1994.

[83] Macario Polo, Ignacio García-Rodríguez, and Mario Piattini. An mda-based approach for database re-engineering. *J. Softw. Maint. Evol.*, 19(6):383–417, November 2007.

[84] D.H. Akehurst, B. Bordbar, P.J. Rodgers, and N.T.G. Dalgliesh. Automatic Normalisation via Metamodelling. In *ASE 2002 Workshop on Declarative Meta Programming to Support Software Development*, September 2002.

[85] Ricardo Perez-Castillo, Ignacio Garcia-Rodriguez de Guzman, Mario Piattini, and Christof Ebert. Reengineering technologies. *IEEE Software*, 28(6): 13–17, 2011. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2011.145.

[86] OMG. Business Process Model and Notation (BPMN). Object Management Group, formal/2011-01-03, 2011. URL http://www.omg.org/spec/BPMN/2.0.

[87] Reda Bendraou, Jean-Marc Jezéquél, and Franck Fleurey. Achieving process modeling and execution through the combination of aspect and model-driven engineering approaches. *Journal of Software: Evolution and Process*, 24(7): 765–781, 2012.

[88] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278. Springer-Verlag, 2005.

[89] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, and I. Trickovic. WS-BPEL Extension for People: BPEL4People. IBM Corporation, http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/, 2005.

[90] OMG. Semantics Of A Foundational Subset For Executable UML Models (FUML), Version 1.0, February 2011. URL http://www.omg.org/spec/FUML/1.0/.

[91] Ralf Ellner, Samir Al-Hilank, Martin Jung, Detlef Kips, and Michael Philippsen. An integrated tool chain for software process modeling and execution. In *Modelling Foundations and Applications*, Lecture Notes in Computer Science, pages 116–131. Springer Berlin Heidelberg, 2012.

[92] TC OASIS. Web Services Business Process Execution Language Version 2.0, April 2007.

[93] Erwan Breton and Jean Bézivin. Model driven process engineering. In *COMPSAC*, pages 225–, 2001.

[94] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008.

[95] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+: A fully configurable multi-user and multi-tool case and came environment. In *CAiSE*, pages 1–21, 1996.

[96] Mario Cervera, Manoli Albert, Victoria Torres, and Vicente Pelechano. A methodological framework and software infrastructure for the construction of software production methods. In Jürgen Münch, Ye Yang, and Wilhelm Schäfer, editors, *New Modeling Concepts for Today's Software Processes*, volume 6195 of *Lecture Notes in Computer Science*, pages 112–125. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-14346-5.

[97] Vicente Pelechano. Automating the development of information systems with the MOSKitt open source tool. In Colette Rolland, Jaelson Castro, and Oscar Pastor, editors, *RCIS*, pages 1–3. IEEE, 2012. ISBN 978-1-4577-1938-7.

[98] Leonardo Mariani and Fabrizio Pastore. Mash: tool integration made easy. *Softw. Pract. Exper.*, 43(4):419–433, 2013.

[99] M. N. Wicks and R. G. Dewar. A new research agenda for tool integration. *Journal of Systems and Software*, 80(9):1569–1585, 2007.

[100] I Kurtev, J Bézivin, and M Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[101] Javier Luis Cánovas Izquierdo, Frédéric Jouault, Jordi Cabot, and Jesús García Molina. Api2mol: Automating the building of bridges between apis and model-driven engineering. *Information and Software Technology*, 54(3): 257–273, 2012.

[102] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Model-driven tool interoperability: An application in bug tracking. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *LNCS*, pages 863–881. Springer, 2006. ISBN 3-540-48287-3.

[103] Yu Sun, Zekai Demirezen, Frédéric Jouault, Robert Tairas, and Jeff Gray. A model engineering approach to tool interoperability. In *Software Language Engineering SLE, Toulouse, France, September 29-30, 2008*, pages 178–187.

[104] C. Amelunxen, F. Klar, A. Konigs, T. Rotschke, and A. Schurr. Metamodel-based tool integration with moflon. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 807–810, May 2008.

[105] Ø'scar Sánchez, Fernando Molina, Jesús García-Molina, and Ambrosio Toval. Modelsec: A generative architecture for model-driven security. *J.UCS*, 15 (15):2957, 2009.

[106] Fernando Wanderley and João Araújo. Generating goal-oriented models from creative requirements using model driven engineering. In *International Workshop on MoDRE, Rio de Janeiro, Brasil, July 15, 2013*, pages 1–9.

[107] J.-H. Jahnke and J. P. Wadsack. Varlet: Human-centered tool support for database reengineering. In *Proceedings of Workshop on Software-Reengineering (WCRE'1999)*, May 1999.

[108] Loup Meurice, Fco Javier Bermudez Ruiz, Jens H. Weber, and Anthony Cleve. Establishing referential integrity in legacy information systems - reality bites! In *ICSME'14*, pages 461–465, 2014.

[109] Jean-Luc Hainaut, Jean Henrard, Jean-Marc Hick, Didier Roland, and Vincent Englebert. The nature of data reverse engineering. In *Proceedings of the 2000 Data Reverse Engineering Workshop (DRE'2000), 2000*, pages –. Zurich Univ. Publish., 2000.

[110] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Softw.*, 24(5):48–55, 2007.

[111] Jennifer Ruttan. *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*, chapter 16: OSCAR. June 2012.

[112] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[113] Jean Henrard, Vincent Englebert, Jan-Marc Hick, Didier Roland, and Jean-Luc Hainaut. Program understanding in databases reverse engineering. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, DEXA '98, pages 70–79. Springer-Verlag, 1998. ISBN 3-540-64950-6.

[114] Eclipse. Cdo model repository. http://www.eclipse.org/cdo/, April 2013. URL `http://www.eclipse.org/cdo/`.

[115] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.

[116] Markus Volter. Md* best practices. *Journal of Object Technology*, 8(6): 79–102, 2009.

[117] Jesús Sánchez Cuadrado, Javier Cánovas, and Jesús García Molina. Applying model-driven engineering in small software enterprises. *Science of Computer Programming*, 2013.

[118] Eclipse-Project. Xtext user guide. http://eclipse.org/Xtext, April 2008. URL `http://eclipse.org/Xtext`.

[119] M. A. Jeusfeld and U. A. Johnen. An executable meta model for re-engineering of database schemas. In *Proceedings of Conference on the Entity-Relationship Approach*, pages 533–547, Manchester, December 1994.

[120] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, June 2008.

[121] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-69927-9_4.

[122] Jesús García-Molina Jesus Sánchez Cuadrado, Javier Cánovas. *Comparison Between Internal and External DSLs via RubyTL and Gra2MoL*. IGI Global, 2012.

[123] José Ramón Hoyos, Jesús García Molina, and Juan A. Botía. A domain-specific language for context modeling in context-aware systems. *Journal of Systems and Software*, 86(11):2890–2905, 2013.

[124] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634. IEEE Computer Society, 2004.

[125] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[126] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, pages 1–17, 2013.

[127] Milan Milanović, Dragan Gašević, Adrian Giurca, Gerd Wagner, and Vladan Devedžić. Bridging concrete and abstract syntaxes in model-driven engineering: a case of rule languages. *Softw. Pract. Exper.*, 39(16):1313–1346, November 2009.

[128] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[129] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez. RubyTL: A practical, extensible transformation language. In *2nd European Conference on Model-Driven Architecture*, volume 4066 of *LNCS*, pages 158–172. Springer, 2006.

[130] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

[131] Markus Völter. MD*/DSL best practices (version 2.0). http://voelter.de/data/pub/DSLBestPractices-2011Update.pdf, April .

[132] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1-2):3–50, 1993.

[133] Axel van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *International Conference on Software Engineering*, pages 5–19, 2000.

[134] Peter Feiler and Watts Humphrey. Software process development and enactment: Concepts and definitions. Technical report, Software Engineering Institute, Carnegie Mellon University, 1992.

[135] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled*. Addison-Wesley, 2012.

[136] Diego Sevilla, Severino Morales, and Jesus Garcia Molina. Inferring versioned schemas from nosql databases and its applications. In *Proc. International Conference on Conceptual Modeling ER*, pages –, 2015.

Colophon

MORE THAN FIVE YEARS OF WORK are now reduced in only 250 pages where I have tried to show the research effort I carried out, sometimes better than others, along with my teaching obligations and a family life. There have been lots of hours addressing new research lines, writing communications and tackling endless reviews. And finally, the hard work provides the expected results and I would asseverate that, by somehow, I distill a research maturity in my professional profile.
Once, somebody told me that in reaching the PhD degree, a researcher should become an expert on a research field, and contribute with his work to the growth of the knowledge of the area. I just hope that my work had reached this achievement and, even though by an infinitesimal piece of knowledge, my contributions had accomplished the appropriate expectations for a PhD.