Data Integration between Objectiver and DB-Main: A Case Study of a Model-Driven Interoperability Bridge

Francisco Javier Bermúdez Ruiz[1], Jesús Joaquín García Molina[1] and Oscar Díaz García[2]
[1]*University of Murcia, Murcia, Spain*
[2]*University of the Basque Country, San Sebastián, Spain*

Keywords: Tool Integration, Interoperability, Model-Driven Engineering, Data Engineering, Goal-oriented Requirements Engineering.

Abstract: In building software systems, the integration of tools with the purpose of exchanging data (i.e. tool interoperability) is common practice. Such an integration is one of the application scenarios of Model-Driven Engineering (MDE), which is often called Model-Driven Interoperability (MDI). In the last few years, some MDI approaches have been presented, and they have shown how MDE techniques are useful in bridging tools in order to integrate data. However, the number of case studies is still limited and more practical experiences of building MDI bridges should be published. In this article, we present an MDI bidirectional bridge that integrates the Objectiver and DB-Main tools. DB-Main database schemas are obtained from Objectiver object models, and they are kept consistent. Through this case study, we contrast the majority of techniques that can be used to implement a MDI solution. We mainly focus on the level of automation offered by each alternative. Some lessons learned are commented on.

1 INTRODUCTION

The development of software systems commonly involves the need to integrate tools. A kind of integration is the exchange of data between tools (Thomas and Nejmeh, 1992). 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 (Geraci, 1991) (Thomas and Nejmeh, 1992). 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 metamodeling and model transformation, are appropriate to implement such bridges, and Model-Driven Interoperability (MDI) is one of the recognized application scenarios of MDE (Brambilla et al., 2012).

Several approaches to building MDI bridges have been presented in the last few years (Bruneliere et al., 2010) (Sun et al.,) (Diallo et al., 2013), which are based on (i) the use of metamodels to represent the information exchanged by each tool, and (ii) the application of a model transformation in order to align the tools (i.e. a mapping between the metamodels), so that the information exchanged can be used in the target tool. Moreover, some examples of MDI bridges between software tools have been described, e.g. code clone detection textual reports to SVG code (Sun et al.,) and Eclipse/EMF (Steinberg et al., 2009) model/metamodels to Microsoft modeling tools (Bruneliere et al., 2010). However, the number of case studies and practical experiences reported is still limited and the presented works have not faced the choice between several implementation alternatives.

In this article we present an MDI bidirectional bridge between the DB-Main and Objectiver tools. DB-Main is a data engineering tool that assists developers in designing database schemas and tasks involved in data reverse engineering, data evolution and data maintenance. Objectiver is a requirement engineering tool based on the KAOS methodology. The bridge aims to obtain DB-Main database schemas from KAOS object models created in Objectiver, as well as the opposite transformation. Therefore, object models and schemas are kept consistent by this bridge. Since DB-Main offers three ways to interoperate (API, XML and grammar format), several MDE techniques may be used to build the syntactic mapping (i.e. obtaining models from exchanged data and vice versa). The different implementation strategies

Ruiz, F., Molina, J. and García, O.
Data Integration between Objectiver and DB-Main: A Case Study of a Model-Driven Interoperability Bridge.
DOI: 10.5220/0005653504770488
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 477-488
ISBN: 978-989-758-168-7

are described and contrasted. With regard to the semantic mapping (i.e. establishing links between the data used in each tool), we have explored the use of QVT relational, and more specifically its capability to write bidirectional model transformations and to keep the source and target models synchronized. To our knowledge this work is one of the first reports on a bidirectional MDI bridge that provides an assessment of different implementation strategies.

The rest of this article is organized as follows. The next section motivates the problem and Section 3 outlines the proposed approach. The implementation of the approach is then described in detail. Firstly, Section 4 introduces the pivot metamodels; Section 5 discusses several strategies to implement the syntactic mapping and includes an assessment of them; and Section 6 analyzes how the semantic mapping can be implemented by means of a bidirectional model transformation written in the QVT Relational language. Section 7 shows the application of the described approach to an object model example. Finally, some related works are commented on and some lessons learned are indicated in the conclusions.

2 DEFINITION OF THE PROBLEM

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 (Thomas and Nejmeh, 1992). 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 (Dardenne et al., 1993). The basic concepts of this method are the following: (i) requirements are described by 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) behavior 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 (Steinberg et al., 2009). 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.

The aim of our work has been to build an MDE interoperability bridge between Objectiver and DB-Main. 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 synchronized.

3 OVERVIEW OF THE APPROACH

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

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

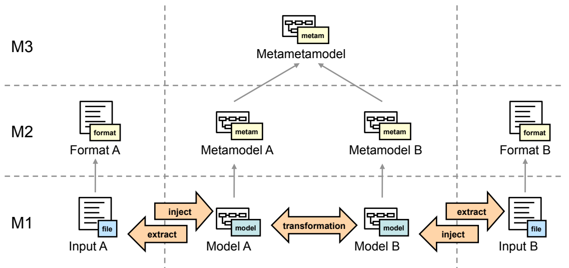


Figure 1: Using MDI techniques between two systems (extracted from (Brambilla et al., 2012)).

through the use of metamodeling 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 (Brambilla et al., 2012). Model transformations ease the implementation of the necessary mappings. In addition, existing MDE tooling can automate some implementation tasks.

Figure 1 shows the elements of a generic MDI bridge for tools A and B (Brambilla et al., 2012), 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 1, 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 metamodeling language (e.g. Ecore). If the metamodels have been created with different metamodeling 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 (Bruneliere et al., 2010). 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 analyzed in Section 5.
- *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 (Stevens, 2013).
- *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 analyzed in Section 5.

Figure 2 shows the Objectiver/DB-Main bridge which has been built so as to achieve MDE interoperability between DB-Main and Objectiver, where database schemas and KAOS object models is 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 kind of injectors

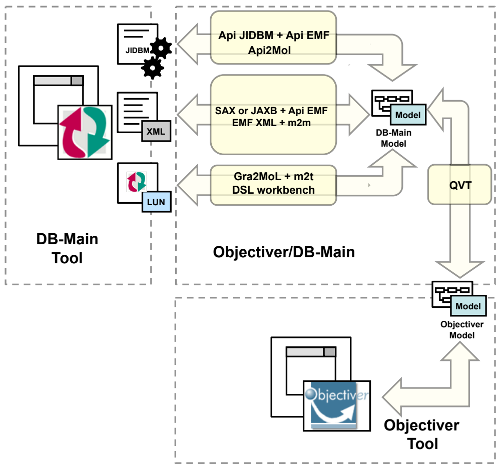


Figure 2: Objectiver/DB-Main bridge.

have been created for DB-Main, taking into account the three options provided by this tool which it uses to access its information. As shown in Figure 2, for each option two implementation strategies have been considered. QVT Relational has been used to implement the semantic mapping.

4 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 3 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*. Packages are composed of KAOS diagrams (*KDiagram*), KAOS entities (*KEntity*) and KAOS relationships (*KRelationship*). KAOS Diagrams provide graphical data in order to visualize 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 characterized by attributes (*Atribute*) which contain a *name* and a *domain*.

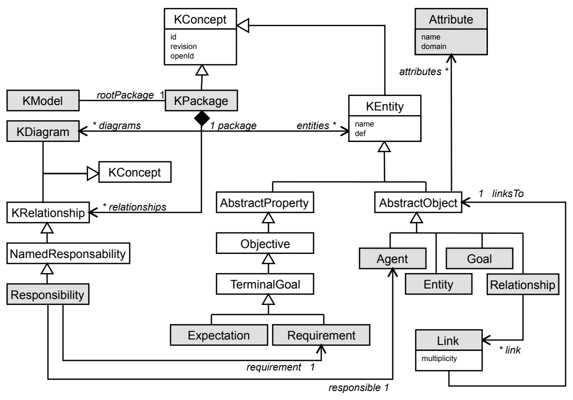


Figure 3: Excerpt of the Objectiver pivot metamodel.

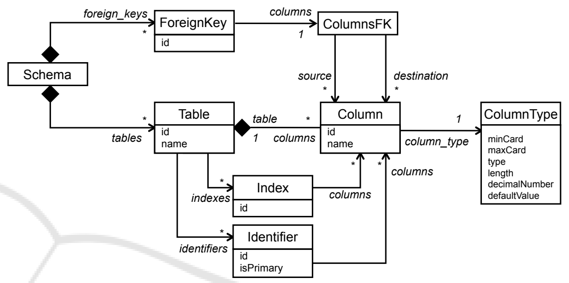


Figure 4: DB-Main pivot metamodel.

We have defined the DB-Main metamodel shown in Figure 4. 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*) 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.

5 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 (Cánovas Izquierdo and García Molina, 2014), 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 (Izquierdo et al., 2012), 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 Section 2, DB-Main provides three alternatives for integrating data: the JIDBM API, and the LUN and XML formats. Figure 5 shows four diagrams which outline the injection/extraction strategies that can be implemented for DB-Main, which are explained below.

5.1 Strategies to Implement the Injection

For the JIDBM API (see Figure 5(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 re-

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

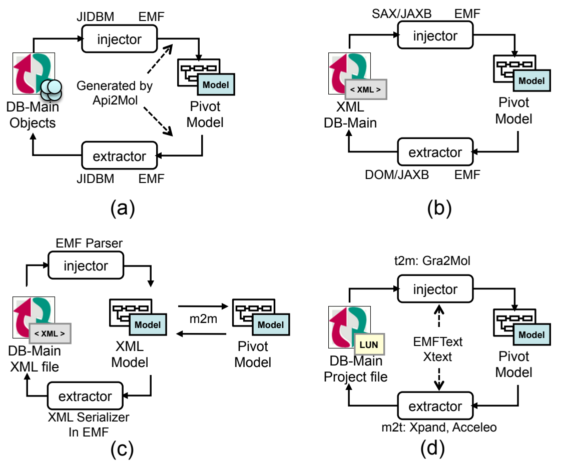


Figure 5: Injectors and extractors for DB-Main.

quired 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 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 5(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 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

[5]http://jcp.org/en/jsr/detail?id=63
[6]http://jcp.org/en/jsr/detail?id=222

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 analyze 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 5(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.

5.2 Strategies to Implement the Extraction

As with injectors, we have experimented with several strategies for each DB-Main interoperability option (see Figure 5). In the case of JIDBM (Figure 5(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.

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 5(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 serialize the XML model into a XML document (Figure 5(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 5(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.

5.3 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 1 summarizes 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.

Table 1: Assessment of the strategies.

	Strategy	Automation	Bidirectional	Easiness	Maturity
API JIDBM	EMF	✗	✗	✓	✓
	API2MoL	✓✓✓	✓	✗	✗
XML Document	SAX/DOM or JAXB + EMF	✗	✗	✓	✓
	EMF XML + m2m	✓✓	✓	✓	✓
LUN format	manual parser/generator + EMF	✗	✗	✓	✓
	Gra2MoL + m2t	✓	✗	✗	✗
	DSL workbench	✓✓✓	✓	✓	✓

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 analyze 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 (Izquierdo et al., 2012). 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 (Cánovas Izquierdo and García Molina, 2014). 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.

6 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 4 and the excerpt of the Objectiver metamodel shown in Figure 3. 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 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” (Stevens, 2013). Below, we describe the defined QVT transformation.

Figure 6 represents the mapping applied without considering foreign keys (FK). The graphical notation depicts the metamodel's classes as boxes and the

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

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

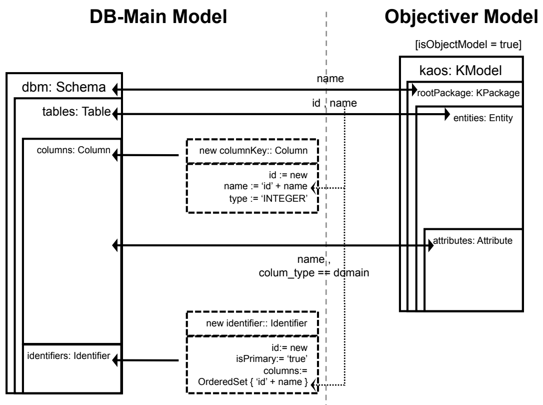


Figure 6: Semantic mapping for $tables \leftrightarrow entities$.

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 6 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 7 represents how FKs are mapped to relationships. This mapping is 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

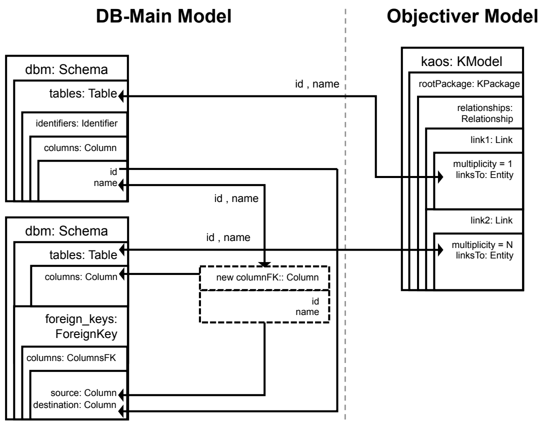


Figure 7: Semantic mapping for $foreign\ keys \leftrightarrow relationships$.

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 7, *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.

```
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.

```
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);
 }
}
```

7 APPLYING THE BRIDGE

We have taken an excerpt of the automated train control system example presented in (van Lamsweerde, 2000) 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 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`.

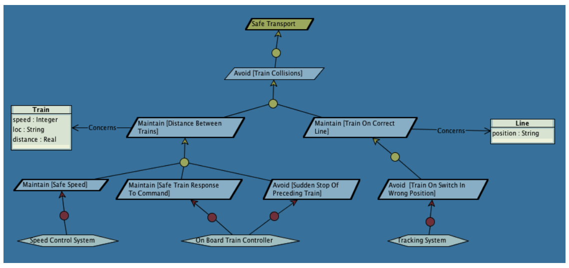

Safe Transport
Avoid [Train Collisions]
Train
speed : Integer
loc : String
distance : Real
Concerns
Maintain [Distance Between Trains]
Maintain [Train On Correct Line]
Concerns
Line
position : String
Maintain [Safe Speed]
Maintain [Safe Train Response To Command]
Avoid [Sudden Stop Of Preceding Train]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

Figure 8: Goal model of the running example.

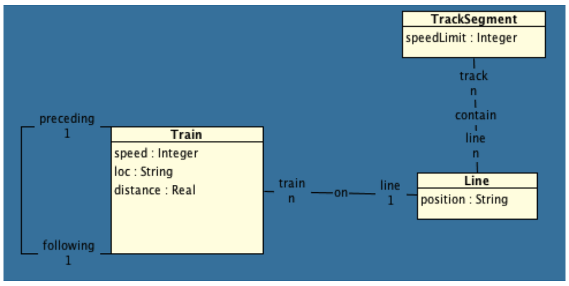
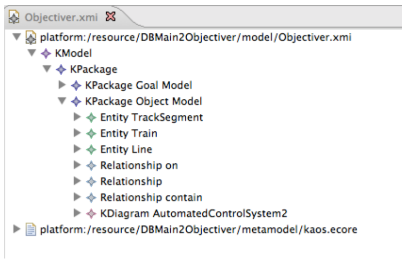


Figure 9: Object model of the running example.

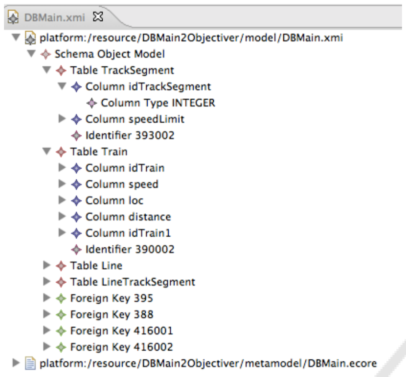
The object model is shown in Figure 9. As can be observed in this figure, the notation for these models complies with those used in UML for class diagrams. This model represents the entities (`Train`, `Line`, and `TrackSegment`) and the relationships between entities, but agents are not included. `Train` has its own 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 10). We should then apply the previously presented QVT transformation to obtain the DB-Main model (see Figure 11). Next, we have performed the API2MoL extractor to generate the DB-Main



```
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 10: 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 11: Ecore model of DB-Main after the semantic mapping.

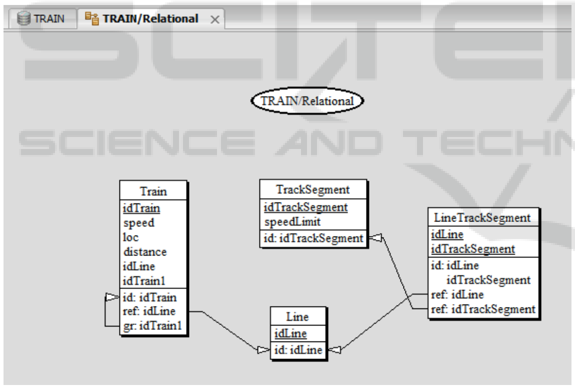
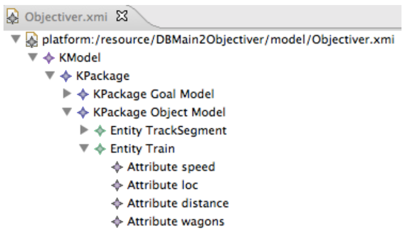


Figure 12: DB-Main project extracted by the syntactic mapping.

database schema, which is shown in Figure 12. 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 synchronization. For instance, we added a new number attribute 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 13 was generated.



```
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 13: Ecore model of the Objectiver project after the modification.

8 RELATED WORK

Tool integration has been a topic of great interest from the early years of software engineering. In (Thomas and Nejmeh, 1992), tool integration is analyzed from several dimensions and more recently a research agenda has been proposed in (Wicks and Dewar, 2007). According to the definitions proposed in (Thomas and Nejmeh, 1992), our work is focused on *tool interoperability* In the review of the literature presented in (Wicks and Dewar, 2007), 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 (Kurtev et al., 2002) to define technologies at a high-level of abstraction. This notion was used in (Bézivin and Kurtev, 2005) to establish bridges between different technologies (e.g. Grammarware, Modelware and Ontologies). Each technical space is characterized 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 metamodeling language for Modelware. In this way, mapping between technologies can be established at three different levels: data, format and formalism. Gra2MoL (Cánovas Izquierdo and García Molina, 2014) and Api2MoL (Izquierdo et al., 2012) 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 (Steinberg et al., 2009). 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.

How MDE could be used in the tool interoperability scenario was shown in (Fabro et al.,) for the AMMA MDE framework. This work proposed to 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 (Sun et al.,), 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 integrate tools is described in (Amelunxen et al., 2008) for the MOFLON framework, which used MOF as metamodeling 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 metamodeling languages, in particular Ecore and metamodeling language used in some Microsoft modeling tools (e.g. DSL Tools) is proposed in (Bruneliere et al., 2010). This approach could be used in our work if a platform different to EMF is used to integrate Objectiver to other tools.

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 model for data warehouses are generated from i* goal models in (Mazón and Trujillo, 2007), security software artifacts (e.g. rules implementing security policies or security code for a database from security models in (Sánchez et al., 2009), or KAOS goal models from mind maps in (Wanderley and Araújo,). 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.

9 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 metamodeling 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 5 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/serializer 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.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support provided by the Objectiver team from the Respect-IT company through one professional license of the Objectiver tool. We also would like acknowledge the Precise research group from the University of Namur their assistance in using DB-Main.

REFERENCES

Amelunxen, C., Klar, F., Konigs, A., Rotschke, T., and Schurr, A. (2008). Metamodel-based tool integration with moflon. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 807–810.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Bruneliere, H., Cabot, J., Clasen, C., Jouault, F., and Bézivin, J. (2010). Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools. In *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, pages 32–47.

Bézivin, J. and Kurtev, I. (2005). Model-based technology integration with the technical space concept. In *Procs. of the Metainformatics Symposium*. Springer-Verlag.

Cánovas Izquierdo, J. and García Molina, J. (2014). Extracting models from source code in software modernization. *Software & Systems Modeling*, 13(2):713–734.

Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-directed requirements acquisition. *Science of computer programming*, 20(1-2):3–50.

Diallo, P. I., Champeau, J., and Lagadec, L. (2013). A model-driven approach to enhance tool interoperability using the theory of models of computation. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, pages 218–237.

Fabro, M. D. D., Bézivin, J., and Valduriez, P. Model-driven tool interoperability: An application in bug tracking. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I*, pages 863–881.

Geraci, A. (1991). *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA.

Izquierdo, J. L. C., Jouault, F., Cabot, J., and Molina, J. G. (2012). Api2mol: Automating the building of bridges between apis and model-driven engineering. *Information & Software Technology*, 54(3):257–273.

Kurtev, I., Bézivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*.

Mazón, J.-N. and Trujillo, J. (2007). A model driven modernization approach for automatically deriving multi-dimensional models in data warehouses. volume 4801 of *LNCS*, pages 56–71. Springer.

Sánchez, O., Molina, F., García-Molina, J., and Toval, A. (2009). Modelsec: A generative architecture for model-driven security. *J.UCS*, 15(15):2957.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition.

Stevens, P. (2013). A simple game-theoretic approach to checkonly qvt relations. *Software and System Modeling*, 12(1):175–199.

Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., and Gray, J. A model engineering approach to tool interoperability. In *Software Language Engineering SLE, Toulouse, France, September 29-30, 2008*, pages 178–187.

Thomas, I. and Nejmeh, B. A. (1992). Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35.

van Lamsweerde, A. (2000). Requirements engineering in the year 00: a research perspective. In *International Conference on Software Engineering*, pages 5–19.

Wanderley, F. and Araújo, J. 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.

Wicks, M. N. and Dewar, R. G. (2007). A new research agenda for tool integration. *Journal of Systems and Software*, 80(9):1569–1585.