Quality Improvement in Data Models with SL_{FD}-based OCL Constraints

Rosario Baena[1], Roberto Aragón[1], Manuel Enciso[1], Carlos Rossi[1], Pablo Cordero[2] and Ángel Mora[2]
[1]*Research Group in Cooperative Information System, University of Málaga, Málaga, Spain*
[2]*Research Group in Mathematics Applied to Computing, University of Málaga, Málaga, Spain*

Keywords: Model-Driven Engineering, Model Quality, OCL, MDE, Logic, Functional Dependency, Data Model, Design by Contract, Model Refactoring.

Abstract: Software verification and modeling quality are permanent challenges in software development. So, smarter and more cohesive methods for the creation and maintenance of data models without loss of quality are required as model complexity increases in current academic and industrial MDE-based system designs. In-place endogenous model transformations (refactorings) are an efficient and straightforward approach to deal with data model complexity, but ad-hoc and frequent transformations must be performed to maintain model quality. In this paper we explore an alternative method to ensure the quality of data models: correction by contract. We propose a new method for the creation and maintenance of static data models (relational, entity-relationship or class models) with enhanced quality. We will use an executable logic for functional dependencies to characterize data model redundancy and we define a set of OCL constraints to guide the construction and maintenance of the models. We also illustrate this approach with a simplified intermediate metamodel (FDMM) for functional dependencies over a data model to show the potential benefits of the method.

1 INTRODUCTION

Static models are at the main core of Model-driven Engineering (MDE) allowing the creation of abstract representations of knowledge associated with any given domain. Transformation of models allows model refactoring to "improve their internal structure without changing their observable behaviour" (Brambilla et al., 2012, Chapter 8). (Mens and Gorp, 2006) proposes a taxonomy of model transformations "that allows us to group tools, techniques or formalisms for model transformation based on their common qualities". Using this taxonomy, we find that we need an *endogenous in-place automated transformation* that allows us to improve some model features or to reduce syntactic complexity by refactoring, among other uses.

Nevertheless, to achieve this goal, newer model transformations have to be defined for every new metamodel and the maintenance of the quality of the models require newer refactorings as soon as these models evolve or become more complex.

To prevent this, we propose the creation of models *correct* by contract. In other words, we establish the necessary restrictions on the metamodel to allow the creation of models *correct* from the beginning, so there is no need to execute any endogenous transformation (i.e. *refactoring*) to maintain the quality of these models after their creation.

Design by contract is a general software design approach first presented in (Meyer, 1997, Chapter 11). It prescribes that a designer should define the formal and precise specification to be satisfied by software components by means of the definition of preconditions, postconditions and invariants, i.e. a *contract*. Analogously to the creation of *classes*, we can define how model elements creation and maintenance can be restricted to be *correct* with a similar contract, as we explain in detail in Section 3.1.

Considering that software components in MDE are represented by their metamodels, the contracts are defined on the metamodel level using OCL constraints. The set of OCL constraints of the contract allows us to validate the correction of some aspects of the models.

We propose a contract in a metamodel that is based on the concept of *functional dependency*, the core concept in the definition of the relational model.

To achieve an optimized model, the contract is guided by a set of equivalences based on inference rules of a functional dependency logic that allows us to prevent data redundancy to be introduced in the cre-

Baena R., Aragón R., Enciso M., Rossi C., Cordero P. and Mora Á..
Quality Improvement in Data Models with SLFD-based OCL Constraints.
DOI: 10.5220/0004593405630569
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 563-569
ISBN: 978-989-8565-68-6

ation of the data model. Therefore, we propose the use of the *Simplification Logic for Functional Dependencies* (Cordero et al., 2002), whose inference rules allow the design of efficient automated methods.

This paper is structured as follows. In section 2 we make a brief introduction to the *Simplification Logic for Functional Dependencies* ($\mathbf{SL}_{\text{FD}}$) and present the equivalences that inspire the constraints that are the basis of the contract that we propose. Section 3 introduces the metamodel for functional dependencies and the specification of the design contract based on the $\mathbf{SL}_{\text{FD}}$ logic by means of the definition of OCL expressions over the metamodel. Finally, we end with the conclusions and future works section 4.

1.1 Related works

There are very few proposals dealing with the concept of functional dependency in the context of MDE. Nevertheless, there is much work where OCL is used to describe some kind of *contract* to the creation and maintenance of properties in models.

An outstanding precedent in the use of OCL as a contract language may be found in (Clavel et al., 2009) where the *unsatisfiability* of OCL invariants, pre and postconditions are introduced as "a powerful tool" for automated reasoning tools. By the other side, in (Siikarla et al., 2004), the declarative and specificative power of OCL are studied in deep. The joint use of pre and postconditions and invariants was originally proposed in (Lano and Kolahdouz-Rahimi, 2012), where model transformation is defined as an UML use case with some logic predicates controlling it. Our approach is based on all these works, but we are focused on model creation and maintenance of properties instead of model transformations.

Finally, closer to our approach, we may cite (Akehurst et al., 2002) which describes an automated method for the normalization of database systems through the creation of an UML profile for database modeling "to encode the definitions of the four normal forms" and a transformation rule "for converting a data model from one normal form to a higher normal form". This approach relies on model transformations whenever a normal form should be reached. In our approach, no transformation is needed. Models are always simplified from construction or invalidated by means of logic constraints over the model, so that the user is warned on where the redundancy is and advised on how it can be avoided.

2 SIMPLIFICATION LOGIC

The *Simplification Logic for Functional Dependencies*, denoted as $\mathbf{SL}_{\text{FD}}$, was first introduced in (Cordero et al., 2002). This logic is equivalent to Armstrong's Axioms (Armstrong, 1974) which are strongly based on the transitivity paradigm. Transitivity is the main obstacle to design efficient automated methods directly based on the inference system.

$\mathbf{SL}_{\text{FD}}$ was designed with the idea of removing redundant attributes and the main inspiration was to replace the transitivity rule by a simplification rule that allows the design of automated methods.

Before describing the inference system of this logic, let us define some preliminary concepts necessary for its understanding. We begin by introducing the notion of functional dependency (FD), which captures a relationship between two set of attributes such that if two tuples agree on attributes of the first set, then they also agree in the second one.

Definition 1. *Let $\mathcal{A}$ be a set of attributes and let $X, Y \subseteq \mathcal{A}$. We say that a relation R satisfies the* **functional dependency** *$X \mapsto Y$ if, for all $t, t' \in R$ we have that: $t_X = t'_X$ implies that $t_Y = t'_Y$.*

The kernel of the Simplification Logic is its novel sound and complete axiomatic system introduced as follows:

Definition 2. *The axiomatic system has one axiom scheme:*

$$\lfloor \texttt{Ax}_{\texttt{FD}} \rfloor \quad \vdash X \mapsto Y, \text{ where } Y \subseteq X$$

The inference rules are the following:

$$\lfloor \texttt{Frag} \rfloor \quad X \mapsto Y \vdash X \mapsto Y' \quad \text{if } Y' \subseteq Y$$
(Fragmentation rule)

$$\lfloor \texttt{Comp} \rfloor \quad X \mapsto Y,\ U \mapsto V \vdash XU \mapsto YV$$
(Composition rule)

$$\lfloor \texttt{Simp} \rfloor \quad X \mapsto Y,\ U \mapsto V \vdash (U - Y) \mapsto (V - Y) \quad \text{if } X \subseteq U \text{ and } X \cap Y = \varnothing$$
(Simplification rule)

Where XY denotes $X \cup Y$ and $X - Y$ denotes the set difference.

The *deduction* ($\vdash$) and *equivalence* ($\equiv$) are defined as usual: We say that a FD φ is deduced from a set of FDs Γ, denoted $\Gamma \vdash \varphi$, if there exists a chain of FDs $\varphi_1 \ldots \varphi_n$ such that $\varphi_n = \varphi$ and, for all $1 \leq i \leq n$, we have that $\varphi_i \in \Gamma$, φ_i is an axiom or is obtained by applying an inference rule to the formulas in $\{\varphi_j \mid 1 \leq j < i\}$.

We say that the sets Γ and Γ' are *equivalent*, denoted $\Gamma \equiv \Gamma'$, if for all FD φ, we have that $\Gamma \vdash \varphi$ if and only if $\Gamma' \vdash \varphi$.

$\mathbf{SL}_{\text{FD}}$ has allowed the design of efficient automated methods: in (Mora et al., 2004) an automated

method to remove redundancy was developed, (Mora et al., 2012) introduces an efficient algorithm based directly on $\mathbf{SL}_{\mathrm{FD}}$ to compute the closure of a set of attributes and finally in (Cordero et al., 2013) a novel algorithm to find all minimal keys based on $\mathbf{SL}_{\mathrm{FD}}$ has been presented.

The design of these methods are due to the different orientation of $\mathbf{SL}_{\mathrm{FD}}$ axiomatic system. More specifically, the characteristics of the logic are the following:

- Formulas are reduced FDs, i.e. formulas of the form $X \mapsto Y$ where $X \cap Y = \varnothing$.
- $\mathbf{SL}_{\mathrm{FD}}$ automated methods look for formulas with the maximum set of attributes in the right hand side. While other classical methods exhaustively use fragmentation rule to get unitary FDs, which increase the set of the input, $\mathbf{SL}_{\mathrm{FD}}$ applies union as much as possible to reduce the number of FDs.
- Simplification rule is centered in the elimination of redundant attributes, which provides a rule which allows to remove redundancy inside the FDs. The classical redundancy removal is strongly based on the need to remove the whole FD.

These issues may be introduced by means of a set of equivalences. We remark that all these equivalences, when read from left to right, are a guide to obtain a simpler set of FD from its original set.

Theorem 1. *Let $X \mapsto Y, U \mapsto V$ be two functional dependencies.*

1. $\{X \mapsto V\} \equiv \{X \mapsto (V - X)\}$
2. $\{X \mapsto Y, X \mapsto V\} \equiv \{X \mapsto YV\}$
3. *If $X \subseteq U$ and $X \cap Y = \varnothing$ then*

$$\{X \mapsto Y, U \mapsto V\} \equiv \{X \mapsto Y, (U - Y) \mapsto (V - Y)\}$$

4. *If $X \subseteq UV$ and $X \cap Y = \varnothing$ then*

$$\{X \mapsto Y, U \mapsto V\} \equiv \{X \mapsto Y, U \mapsto (V - Y)\}$$

The above equivalences will be named respectively *Reduction Equivalence* (a particular case of Fragmentation), *Union Equivalence* (a particular case of Composition), *Simplification Equivalence* (based on the Simplification Rule) and *Right Simplification Equivalence* (derived from Simplification Equivalence that allows us to remove redundancy in the right side of the formula).

These four equivalences are the basis of the contract presented in this paper and they will induce a set of OCL constraints (invariants, pre and postconditions) to ensure the quality of the models with no loss of semantics.

3 $\mathbf{SL}_{\mathrm{FD}}$-*simplified* DATA MODEL

In this section, we describe the definition of the contract that will allow us to create $\mathbf{SL}_{\mathrm{FD}}$-*simplified* models. Also, we show a simple example to prove its benefits.

The metamodel of figure 1, based on the metamodel presented in (Enciso et al., 2012), is the basis to the definition of the necessary constraints to validate the correctness (by contract) of the models.

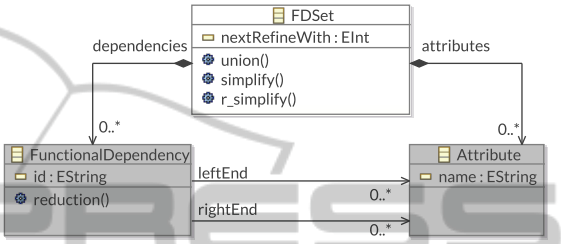

Figure 1: Metamodel for Functional Dependencies (FDMM). Class operations are defined only to maintain the quality of the data model, not to represent executable or simulation behaviour of any implementation derived from the model.

3.1 The Metamodel Contract

Bertrand Meyer, in (Meyer, 1997), define the concept of design by contract as the formal specification of the rights and obligations established between the software components and its clients. The definition of the contract on the software components is done by using two types of assertions: pre/postconditions and invariants. The invariants allow to express the global properties to be satisfied by all instances of a class, while pre and postconditions describe the contract of the class methods.

Analogously, since the software components in a MDE system are represented by metamodels, so the contracts can be defined as constraints on its model elements (invariants) and its model elements operations (pre and postconditions) (Cabot and Gogolla, 2012). In our case, OCL (OMG, 2012) is used to establish the design contract of models conforming to these metamodels.

The violation of an OCL invariant triggers a warning indicating that a model is not in a $\mathbf{SL}_{\mathrm{FD}}$-*simplified* state. The satisfaction of an OCL precondition (of a simplification operation) provides an advice to the user to perform the corresponding optimization. Thus, the OCL postcondition of that particular operation will be satisfied. Note that, if no other preconditions are satisfied, the global state will change to

$\mathbf{SL}_{FD}$-*simplified* and there will be no other violation of invariants.

We have performed a practical proof of concept of this method using the Dresden OCL toolkit (Demuth, 2004) with a comprehensive set of test models with good results. We choose this toolkit because common modeling tools with built-in OCL does not support advanced validation features (like pre and postconditions).

3.2 OCL Invariant

The invariant is defined in the context of the functional dependency element, so that it is possible to identify the element not meeting that constraints.

```
context FunctionalDependency
 inv:
    let X: Set(Attribute) =
           self.leftEnd->asSet() in
    let Y: Set(Attribute) =
           self.rightEnd->asSet() in
(1) Y->intersection(X)->isEmpty()
    and
(2) FunctionalDependency.allInstances()->
    excluding(self)->forAll(uv|
      let U: Set(Attribute) =
             uv.leftEnd->asSet() in
      let V: Set(Attribute) =
             uv.rightEnd->asSet() in
      X <> U and
     (U->includesAll(X) implies
      U->intersection(Y)->isEmpty())
     and
     (U->union(V)->includesAll(X) implies
      V->intersection(Y)->isEmpty()))
```

The first part of the invariant states that both sides of every dependency should have disjoint attribute sets to preserve the *Reduction Equivalence*. The second part of invariant establishes the conditions to address the *Union*, *Simplification* and *Right Simplification Equivalence*. We evaluate each dependency with all the other dependencies of the set by iterating over all instances of functional dependencies (`allInstances`) and excluding the dependency currently being validated (`excluding(self)`).

3.3 OCL Pre and Postconditions

The pre and postconditions guide the application of the corresponding operations on the model elements. More specifically, we define an operation for every equivalence of the theorem 1:

```
context FunctionalDependency
   reduction(): FunctionalDependency
context FDSet
        union(): OrderedSet(FunctionalDependency)
     simplify(): OrderedSet(FunctionalDependency)
   r_simplify(): OrderedSet(FunctionalDependency)
```

Notice that the `reduction` operation is defined within the context of the `FunctionalDependency` element while the rest of operations are defined within the context of `FDSet` element.

Reduction. This operation ensures an empty intersection between the left and right sets of attributes of a functional dependency.

```
context FunctionalDependency::reduction()
  pre:
    let X: Set(Attribute) =
           self.leftEnd->asSet() in
    let Y: Set(Attribute) =
           self.rightEnd->asSet() in
    not Y->intersection(X)->isEmpty()
  post:
    let X: Set(Attribute) =
           self.leftEnd->asSet() in
    let Y: Set(Attribute) =
           self.rightEnd->asSet() in
    Y->intersection(X)->isEmpty()
```

Union. This operation is applied to those dependencies in the total set with the same left hand sides. The postcondition describes that all dependencies whose left hand side are equal each other are now joined together in a single dependency whose right hand side is the result of the union of the original right hand sides.

```
context FDSet::union()
  pre: dependencies->exists(xy, uv|
    let X: Set(Attribute) =
           xy.leftEnd->asSet() in
    let U: Set(Attribute) =
           uv.leftEnd->asSet() in
    xy.id <> uv.id and X = U
  )
  post: dependencies->forAll(xy|
    let X: Set(Attribute) =
           xy.leftEnd->asSet() in
    let Y: Set(Attribute) =
           xy.rightEnd->asSet() in
    dependencies@pre->forAll(uv| -- (note 1)
      let U: Set(Attribute) =
             uv.leftEnd->asSet() in
      let V: Set(Attribute) =
             uv.rightEnd->asSet() in
      X = U implies Y->includesAll(V)))
```

Simplify. This operation reduces a given dependency by removing attributes on both sides of the dependency if there exists another dependency such that its left hand side is a subset of the left hand side of the target dependency.

```
context FDSet::simplify()
  pre: self.dependencies->exists(xy, uv|
    let X: Set(Attribute) =
```

[1] @pre: refer to the value of a feature before the operation execution (in the state checked in the precondition).

```
        xy.leftEnd->asSet() in
  let Y: Set(Attribute) =
        xy.rightEnd->asSet() in
  let U: Set(Attribute) =
        uv.leftEnd->asSet() in
  let V: Set(Attribute) =
        uv.rightEnd->asSet() in
  xy.id <> uv.id and
  Y->intersection(X)->isEmpty() and
    U->includesAll(X) and
    (not U->intersection(Y)->isEmpty() or
    not V->intersection(Y)->isEmpty()))
post: self.dependencies->forAll(xy, uv|
  let X: Set(Attribute) =
        xy.leftEnd->asSet() in
  let Y: Set(Attribute) =
        xy.rightEnd->asSet() in
  let U: Set(Attribute) =
        uv.leftEnd->asSet() in
  let V: Set(Attribute) =
        uv.rightEnd->asSet() in
  xy.id <> uv.id and
  Y->intersection(X)->isEmpty() and
    U->includesAll(X) and
    U->intersection(Y)->isEmpty() and
    V->intersection(Y)->isEmpty())
```

R-Simplify. This operation extends the simplify rule when the above operation cannot be applied because the subset inclusion does not fulfills between the two left had sides but the inclusion is validated between the left had side of a given dependency and all the attributes of the target one. In this operation, attributes are removed only in the right hand side of the target dependency.

```
context FDSet::r_simplify()
  pre: self.dependencies->exists(xy, uv|
    let X: Set(Attribute) =
          xy.leftEnd->asSet() in
    let Y: Set(Attribute) =
          xy.rightEnd->asSet() in
    let U: Set(Attribute) =
          uv.leftEnd->asSet() in
    let V: Set(Attribute) =
          uv.rightEnd->asSet() in
    xy.id <> uv.id and
    Y->intersection(X)->isEmpty() and
      U->union(V)->includesAll(X) and
      not V->intersection(Y)->isEmpty())
  post: self.dependencies->forAll(xy, uv|
    let X: Set(Attribute) =
          xy.leftEnd->asSet() in
    let Y: Set(Attribute) =
          xy.rightEnd->asSet() in
    let U: Set(Attribute) =
          uv.leftEnd->asSet() in
    let V: Set(Attribute) =
          uv.rightEnd->asSet() in
    xy.id <> uv.id and
    Y->intersection(X)->isEmpty() and
      U->union(V)->includesAll(X) and
      V->intersection(Y)->isEmpty())
```

3.4 An Illustrative Example

To illustrate the benefits of our approach, we show how the OCL constraints (the contract) guide the creation and evolution of a model of functional dependencies based on the FDMM (fig.1).

We consider the example presented in (Ullman and Widom, 1997, Chapter 3) that describes the relation `Movies` as follows:

title	year	studioName	starName
Start Wars	1977	Fox	Carrier Fisher
Start Wars	1977	Fox	Mark Hamill
Start Wars	1977	Fox	Harrison Ford
Mighty Ducks	1991	Disney	Emilio Estevez
Wayne's World	1992	Paramount	Dana Carvey
Wayne's World	1992	Paramount	Mike Meyers
Sleepy Hollow	1999	Paramount	Johnny Depp

In the previous data table, the following functional dependency[2] holds:

```
1. starName, title, year -> studioName
```

In a new iteration of the design of the model (made by the same designer or by a cooperator) new attributes are incorporated to the relation: the director of the movie, an indicator saying if the star is the director's *pet actor* and the movie revenues. This new attributes rely on these new functional dependencies:

```
2. starName,title,year -> director, petActor,
                          revenues
3. starName,director -> petActor
4. title,year -> studioName, director
```

When validating the whole set of dependencies we find that the model violates the contract: the invariant is not satisfied by any dependency and the preconditions of the operations `union`, `simplify` and `r_simplify` are satisfied.

To obtain a $\mathbf{SL}_{\mathrm{FD}}$-*simplified* model, we follow the advise to apply these operations:

- The `union` operation is applied on the model, so that dependencies 1 and 2 should be fused in one dependency:

  ```
  starName,title,year -> studioName,director,
                         petActor,revenues
  ```

- The `r_simplify` operation is applied. The dependency obtained from the previous union should be simplified with the dependency 3:

  ```
  starName,title,year -> studioName,director,
                         revenues
  ```

- The `simplify` operation is applied. The dependency obtained from the previous simplification should be simplified with the dependency 4:

[2]We use hereafter an informal notation to improve readability fully interchangeable with its XMI counterpart.

```
starName,title,year -> revenues
```

Now, we get a new model that is $\mathbf{SL}_{FD}$-*simplified* (invariants and postconditions are satisfied, but preconditions are not):

```
1. starName,title,year -> revenues
2. starName,director -> petActor
3. title,year -> studioName,director
```

Notice that this new set has been reduced in the number of dependencies and in the number of attributes inside the dependencies.

4 CONCLUSIONS AND FUTURE WORKS

We have seen that it is possible to create and maintain some quality standards of data models of functional dependencies just by defining a contract on the FDMM. This contract is defined using only OCL expressions embedded in the metamodel and it is used both to validate the global state of the model and to warn about the need to apply simplification operations.

Our contribution in centered in the field of MDE models quality enhancement by reusing the concept of design-by-contract (Meyer, 1997), formal methods from previous research efforts (Enciso et al., 2012; Cordero et al., 2002) and the standard language for the design of smart models, OCL (OMG, 2012). Nevertheless, the approach to correction-by-contract in model creation is new (to our best knowledge) and a promising perspective to be consider in future works.

We plan to extend this method to the most common metamodels for static models like relational, entity-relationship and UML class models. For this, it would be necessary to study the interpretation of the concept of functional dependency in these metamodels and adapt the contract defined for the FDMM accordingly, so that the semantics of the set of functional dependencies are preserved although some syntactical differences may exist.

We have illustrated that, even when there is a reasonably good elicitation of functional dependencies, some redundancy may arise when the model is maintained. In this scenario, our method can help to control the introduction of redundancies and avoiding loss of quality.

The method presented in this paper is strongly based on a set of equivalences that characterizes the strength of the axiomatic system of the $\mathbf{SL}_{FD}$. These equivalences induce a set of OCL constraints (invariants, pre and postconditions) which allow us to ensure the quality of the models while they are created such that no further transformation is required.

ACKNOWLEDGEMENTS

Supported by grant **Andalucía Tech** (International Campus of Excellence), **IPT-2011-15597770000** of the Economy and Competitiveness Ministry and **TIN2011-28084** of the Science and Innovation Ministry of Spain co-funded by the European Regional Development Fund (ERDF).

REFERENCES

Akehurst, D., Bordbar, B., Rodgers, P., and Dalgliesh, N. (2002). Automatic Normalisation via Metamodelling. In *ASE 2002 Workshop on Declarative Meta Programming to Support Software Development*.

Armstrong, W. W. (1974). Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-driven Software Engineering in Practice*. Synthesis digital library of engineering and computer science. Morgan & Claypool Publishers.

Cabot, J. and Gogolla, M. (2012). Object constraint language (OCL): a definitive guide. In *Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering*, SFM'12, pages 58–90, Berlin, Heidelberg. Springer-Verlag.

Clavel, M., Egea, M., and de Dios, M. A. G. (2009). Checking unsatisfiability for ocl constraints. *ECEASST*, 24.

Cordero, P., Enciso, M., and Angel, M. (2013). Automated reasoning to infer all minimal keys. International Joint Conference on Artificial Intelligence, Beijin, China.

Cordero, P., Enciso, M., Mora, A., and Guzmán, I. P. d. (2002). Slfd logic: Elimination of data redundancy in knowledge representation. In *Proceedings of the 8th Ibero-American Conference on AI: Advances in Artificial Intelligence*, IBERAMIA 2002, pages 141–150, London, UK, UK. Springer-Verlag.

Demuth, B. (2004). The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD2004)*.

Enciso, M., Rossi, C., and Guevara, A. (2012). A metamodel for functional dependencies - towards a functional dependency model transformation. In *ICSOFT'12*, pages 291–296.

Lano, K. and Kolahdouz-Rahimi, S. (2012). Constraint-based specification of model transformations. *Journal of Systems and Software*.

Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(0):125 – 142.

Meyer, B. (1997). *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Mora, A., Cordero, P., Enciso, M., Fortes, I., and Aguilera, G. (2012). Closure via functional dependence simplification. *International Journal of Computer Mathematics*, 89(4):510–526.

Mora, n., Enciso, M., Cordero, P., and Prez de Guzmn, I. (2004). An efficient preprocessing transformation for functional dependencies sets based on the substitution paradigm. In Conejo, R., Urretavizcaya, M., and Prez-de-la Cruz, J.-L., editors, *Current Topics in Artificial Intelligence*, volume 3040 of *Lecture Notes in Computer Science*, pages 136–146. Springer Berlin Heidelberg.

OMG (2012). Object Constraint Language (OCL). OMG Standard, v. 2.3.1.

Siikarla, M., Peltonen, J., and Selonen, P. (2004). Combining ocl and programming languages for uml model processing. In *Proceedings of the Workshop, OCL*, volume 2, pages 175–194.

Ullman, J. D. and Widom, J. (1997). *A first course in database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.