Submitted to WOOR 2007

A Meta-model Approach to Inconsistency Management

Anne Keller, Serge Demeyer

Department of Mathematics and Computer Science
University of Antwerp, Belgium
anne.keller,serge.demeyer@ua.ac.be

Abstract

Transformations between abstract design models and platform specific models are the basis of the model-driven development process. In the transformation process inconsistencies are introduced, that neither can nor always should be eradicated. This paper presents a meta-model that addresses the need to manage inconsistencies arising during vertical model transformations. The meta-model captures both characteristics of inconsistencies and provides support for inconsistency management activities such as monitoring and analysis. We motivate the inconsistency management approach with an example transformation and show the scope of the proposed meta-model solution.

1 Introduction

Today's software runs on various platforms during its lifetime. An application might be deployed on a mobile device as well as in a web-based environment. To map the application to a specific platform, technology specific decisions need to be taken and be incorporated into the design. If this platform specific design is mapped onto a different platform a new design, specific to the new platform, is created. To avoid re-designing an application for each platform, model-driven development proposes to decouple the abstract design of an application from its platform specific design. Thus, the model-driven development process addresses the need to adapt the software development process to the mapping of software design models onto various platforms.

In the model-driven development process a platform and implementation independent design model (PIM) is created, which serves as the basis of information about the application. Stepwise applied transformations map this abstract design model onto different target platforms. The resulting target models (PSMs) incorporate all implementation specific details. [2]

In order to transform a PIM into a PSM a modeler applies a set of transformation rules, which describe how each model element is adapted (i.e., changed, deleted, created). The modeler chooses the correct rules from a rule catalogue and determines their order, based on his knowledge of the source model and both the source and target meta-model.

While applying transformation rules, the modeler also manages the system's consistency. That is, descriptions created in the target model should not contradict descriptions made in any other part of the system (e.g., in the source model, in the target meta-model, etc.). In this paper we suggest a meta-model to support inconsistency management. [6] The meta-model is able to capture both characteristics of inconsistencies as well as information on how different inconsistency management activities, such as monitoring and analysis, should be supported. This promotes an inconsistency management approach that augments inconsistency detection with additional techniques and allows the modeler to optionally consider inconsistency information during the whole transformation process. This enables the modeler to avoid inconsistencies caused by complex specifications (e.g., large models, on which different sets of consistency rules, that might overlap and possibly be contradicting, are specified), and to flexibly manage the system's consistency (e.g., to tolerate inconsistencies temporarily).

The rest of the paper is structured as follows. In Section 2 we introduce an example transformation together with different types of inconsistencies motivating the meta-model solution. Section 3 introduces the scope of the inconsistency meta-model. And Section 4 concludes with a presentation of future work.

2 Vertical Model Transformations

To transform an abstract model into a platform specific model (i.e., to apply vertical transformations [4]) is a task that requires decisions based on various aspects of the system. The target model should incorporate requirements of the target platform and provide a optimal design solution. Further, design decisions are not fully independent of other platform specific models describing other views of the same application, since these are required to interact and only together compose a complete system.

2.1 Applying Transformation Rules

On the example of *Rosa's Breakfast Service*, introduced by Kleppe et al. [2], we show that applying model transformations is not a self-contained process and confronts the modeler with various decisions.

An excerpt of the PIM of Rosa's Breakfast Service is shown in Figure 1. Through the service a customer can order a breakfast and have it delivered to his home. Orders can be made for any number of people, where everyone can receive a different breakfast. The PIM of Rosa's Breakfast Service is transformed into three PSMs each representing one tier of a three-tier architecture. A database in SQL for the data tier, the middle tier using Enterprise Java Beans (EJB), and the presentation tier using Java Server Pages (JSP).[1]

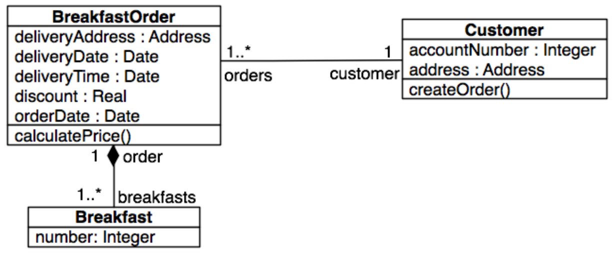

Figure 1. Excerpt of the PIM of Rosa's Breakfast Service

To transform the UML model in Figure 1 into the relational database model in Figure 2, the modeler applies transformation rules. One transformation step is

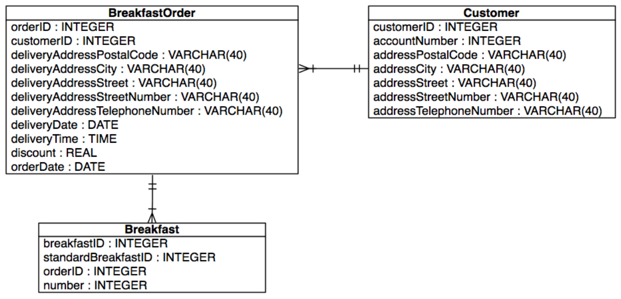
BreakfastOrder
orderID : INTEGER
customerID : INTEGER
deliveryAddressPostalCode : VARCHAR(40)
deliveryAddressCity : VARCHAR(40)
deliveryAddressStreet : VARCHAR(40)
deliveryAddressStreetNumber : VARCHAR(40)
deliveryAddressTelephoneNumber : VARCHAR(40)
deliveryDate : DATE
deliveryTime : TIME
discount : REAL
orderDate : DATE
Customer
customerID : INTEGER
accountNumber : INTEGER
addressPostalCode : VARCHAR(40)
addressCity : VARCHAR(40)
addressStreet : VARCHAR(40)
addressStreetNumber : VARCHAR(40)
addressTelephoneNumber : VARCHAR(40)
Breakfast
breakfastID : INTEGER
standardBreakfastID : INTEGER
orderID : INTEGER
number : INTEGER

Figure 2. Excerpt of the Relational PSM of Rosa's Breakfast Service

to map the data types specified in UML to data types used in a relational database, SQL in this example. The mapping is mostly straight forward and includes rules such as:

- A UML String is mapped onto a SQL VARCHAR(40),
- A UML integer is mapped onto a SQL INTEGER.

For example, *AccountNumber : Integer* in class *Customer* is transformed to *AccountNumber : INTEGER* in entity *Customer*.

Less straight forward is the mapping of the data type *Address*. The modeler is given two choices to map the data type *Address* in *deliveryAddress : Address* to a relational expression. One possibility is to create a new table for this data type. Another option is to include the data type into the table, which represents the class holding the attribute. In the presented example the modeler chooses to include the data type *Address* into the *Customer* entity. Here a column is created for each field in the data type—*adressPostalCode*, *addreesCity*, *addressStreet*, etc. This choice is motivated by the wish to align the relational PSM with the later created EJB PSM, as explained in the following.

Figure 3 shows the top-level component model of the EJB PSM for Rosa's Breakfast Service. To improve performance a coarse grained component model is chosen. This means that a single EJB component is large to decrease the frequency of interactions between components. The trade-off is that a lot of data is communicated in each interaction. To transform the PIM to a EJB PSM each outermost PIM class (i.e., a class that is not composite part of another class) is transformed in a EJB data class. Each PIM attribute of one class is transformed into a EJB attribute mapped to the corresponding EJB data class. According to this the class *Customer* from the PIM is transformed into a

[1]For brevity neither the complete PIM of Rosa's Breakfast Service nor the complete transformations or all PSMs are shown here. Refer to [2] and a demo edition of OptimalJ, in which the example is realized, for further information. *http://frontline.compuware.com/javacentral/demos/26002.asp*

data class *Customer* containing the attributes *accountNumber* and *address*. As in the Relational PSM, the attribute *address* is part of the *Customer* class/entity.

This example shows that the performance of other architectural components is relevant for the transformation of a PIM into a PSM. To choose an optimal solution, the modeler needs to consider the transformation and platform at hand, additionally to other platforms contributing to the complete system. In the following we show that this also holds for managing the system's consistency.

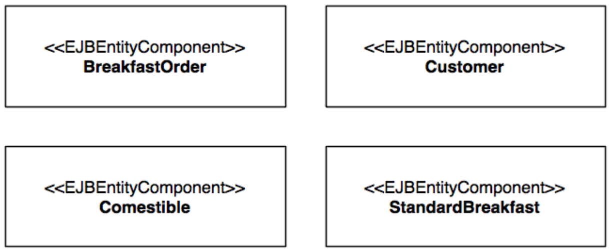

Figure 3. EJB Component Model of Rosa's Breakfast Service

2.2 Inconsistency Management

At this point the modeler has not addressed consistency issues yet. Inconsistencies arise when assertions are made at one point of the system that are contradicted by different assertion at another point in the system. For example, the created target model can be inconsistent with its source model, with its meta-model and with additionally specified consistency rules. In the following these three types of inconsistencies are introduced, to show various issues that an inconsistency management strategy needs to address.

Inconsistency between source and target model arise, for example, through an ambiguously defined source model. Every field in the *Address* data type of the example in Figure 1 is of type String. Every String is mapped onto VARCHAR(40) in the *Customer* entity (Figure 2). In UML it is not specified of how many characters a String should consist. To replace this ambiguously defined value with a fixed value in all cases, creates an inconsistency, if the UML designer intended different string lengths for different fields, e.g., for optimization reasons or to provide more space for data input to a specific field. This inconsistency becomes apparent, if data usage of the system is not as expected or a long too string is input. For example, if a longer string for *addressStreet* is entered, this inconsistency can render the street data useless.

Inconsistency between the target model and its meta-model likely occurs when the modeler overlooks a rule in the meta-model choosing transformation rules. For example, the meta-model in Figure 4 shows an excerpt of a relational meta-model. [1] A relational table contains several columns. Columns have a reference to their type. Tables, columns as well as types have names, shown by the inheritance from *Named*.

The modeler adheres to this meta-model creating transformations. However, additional constraints are specified on the meta-model for a specific relational database. For example, constraints that represent *Access* naming conventions are added. [3] If the modeler overlooks a naming rule, an inconsistency of the target model with its meta-model is created. Additionally the group might be moving from one set of naming conventions to another, therefore also the new set of invariants is added to the meta-model. The modeler now needs to decide, which rule set to choose. An inconsistency is introduced, if he chooses to adhere to the wrong set of invariants or does not adhere to any naming conventions stated.

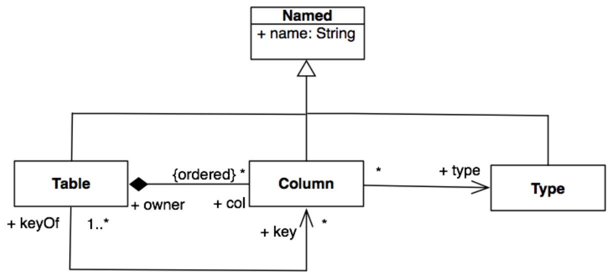

Figure 4. Relational Meta-model

Inconsistencies between the target model and additionally defined consistency rules, addressing issues such as performance, standard compliance, etc., occur when these rules are overlooked or contradict other rules. To transform a PIM into a relational PSM, additional rules might be specified to make the resulting model comply to a relational Normal Form. For example, to comply to 1NF (First Normal Form) a rule to eliminate repeating groups states: 'Make a separate table for each set of related attributes, and give each table a primary key'. [5] In Rosa's Breakfast Service this suggests to keep *BreakfastOrder* and *Customer* as separate tables, in which *deliveryAddress* and *address*, both of type *Address*, are included, instead of creating a separate table for the type *Address*. An inconsistency with 1NF occurs, when this rule is violated.

The examples above show various issues related to inconsistency.

- Different kinds of inconsistencies exist, that are present either explicitly through violated consistency rules or implicitly. Following the examples above, well-formedness rules and rules specified to comply to 1NF are two different kinds of consistency rules, thus creating two different kinds of inconsistencies. However, especially implicitly defined inconsistencies are hard to capture and pose a problem. For example, the intent of the UML modeler creating the *Address* data type is implicitly captured. And thus, an inconsistency with this intent is implicit since no explicit consistency rule is violated.
- Besides from inconsistent specifications, inconsistencies occur through mistakes made by the modeler. Mistakes are easily made, if the modeler deals with large models, with various additionally defined rules. If these rules are overlapping or even contradicting, additional challenges occur.
- Different consistency rules have different importance and different impact. Different importance means that different consistency rules are relevant at different points of the transformation process. For example, rules to comply to 1NF are especially relevant in transformation steps that decide on the structure. While later supposedly more fine grained changes are made and thus different consistency rules become important. The impact of an inconsistency describes what consequences an inconsistency causes. This can range from system crashes, over inconsistent values, to non-conformance to style rules.

3 An inconsistency meta-model

The problem the modeler faces managing the systems consistency is twofold. On the one hand, information about inconsistencies, ranging from a list of consistency rules to additional information such as the impact of an inconsistency on the system, is required. On the other hand, support for different inconsistency management activities, such as monitoring, is needed.

A solution lies in an inconsistency meta-model. An inconsistency meta-model captures information such as

- characteristics of inconsistencies,
- different kinds of inconsistencies,
- indicators for inconsistencies,
- and impact of inconsistencies.

This knowledge is accessible to the modeler and augments the view of the system. The terminology and concepts used in the meta-model can be common ground to communicate about inconsistencies in a group or domain. The instances of the meta-model add to the body of knowledge of different kinds of inconsistencies and makes this information retrievable, exchangeable and extendible.

The information is explicitly and implicitly present in the inconsistency meta-model. Thus, tool support for the meta-model will make this information accessible and support the creation and adherence to an inconsistency management strategy.

Techniques that help to establish an inconsistency management strategy and are supported in the meta-model include:

- **Add inconsistency kinds.** - Adding new inconsistency kinds, i.e., new consistency rule sets, can make implicit inconsistencies explicit. That is, once an implicit inconsistency is uncovered it can be saved for reuse. Also rule sets specific to a domain or community can be added as new inconsistency kind.
- **Set inconsistency kind to check.** - Setting which inconsistency kind is to be checked at one moment in time, pays respect to different inconsistency kinds having different importance. Thus, turning checking for certain rule sets off, helps to clarify the output and gives more attention to the currently important rule sets.
- **Choose strictness in detection.** - Manipulating the strictness of inconsistency checking adheres to the believe that different inconsistency kinds have different impact. For example, strict checking (e.g., display a warning for each violation) is turned on for crucially important rules.
- **Analyze consistency rules.** - Analyzing consistency rules is one possibility to assist the modeler in dealing with large models with contradicting or overlapping rules. Listing all rules specified on one element, or listing all elements affected by one rule are examples.
- **Monitor consistency.** - Monitoring the development of certain model elements can act as a filter for large models. That is, the monitored element is recognized as generally important and thus highlighted. For example, alerts are shown whenever a monitored model element violates a certain consistency rule.

- **Visualization.** - Visualization is another technique that especially supports dealing with large models and many rules. For example, visualizing all model elements to which explicit rules are tied, can convey a sense of hotspots in the models. Or marking all model elements touched by one rule shows the impact a violation of that rule could have.

- **Track history of inconsistencies.** - Recording which inconsistencies arose and how or if they were resolved, can point to model elements that are prone to inconsistencies and should be monitored. Also the information that the violation of a specific consistency rule was tolerated before can trigger a certain decision, e.g., either to tolerated the violation again or to explicitly resolve the violation this time.

4 Future Work

In this paper a meta-model to manage inconsistencies is suggested. The most immediate steps are studying different kinds of inconsistencies to find which are 'natural' ideas and concepts to describe inconsistencies in the vertical model transformation context. Which abstractions are adequate and describe the modelers reality? What concepts unite different kinds of inconsistencies on different platforms? Here also the question whether those kinds of inconsistencies are present in other model transformations is relevant.

Subsequently, studying a transformation case study will provide insight into whether the found characteristics apply to found inconsistencies. The case study will also yield understanding of techniques that are a useful to support in inconsistency management. How does the user access the provided information and during which task does he require the information?

The meta-model incorporating the found information will finally be validated. A suitable way to validate the meta-model and fitting validation criteria will have to be chosen.

References

[1] ATL Transformation Example - Class to Relational. http://www.eclipse.org/m2m/atl/atlTransformations/Class2Relational/ExampleClass2Relational[v00.01].pdf, March 2005.

[2] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[3] Stan Leszynski and Greg Reddick. Naming Conventions for Microsoft Access - The Leszynski/Reddick Guidelines for Microsoft Access. http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnaraccess/html/msdn_20naming.asp, May 1994.

[4] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.

[5] Mike Nicewarner. Rules of Data Normalization. http://www.datamodel.org/NormalizationRules.html, June 2005.

[6] G. Spanoudakis and A. Zisman. *Handbook of Software Engineering and Knowledge Engineering*, chapter Inconsistency Management in Software Engineering: Survey and Open Research Issues, pages 329–380. World Scientific Publishing Co., 2001.