FROM AN ABSTRACT OBJECT-ORIENTED DOMAIN MODEL TO A META-MODEL FOR THE DOMAIN
Model Driven Development of a Manufacturing Execution System

Antoine Schlechter, Guy Simon and Fernand Feltz
Centre de Recherche Public Gabriel Lippmann, rue du Brill, Belvaux, Luxembourg

Keywords: MDE, Model Driven Engineering, MDSD, Model Driven Software Development, Meta-Modeling.

Abstract: Despite a broad agreement on the benefits of model driven approaches to software engineering, the use of such techniques is still not very widespread. We think this is due to the appearing discouraging difficulty of meta-modeling. This paper presents a new method to easily obtain a meta-model from an abstract object-oriented domain model. The method is applied to the development of a Manufacturing Execution System.

1 INTRODUCTION

Since quite a while, model-driven approaches to software engineering such as Model Driven Engineering (MDE), Model Driven Software Development (MDSD) or Model Driven Architecture (MDA) have been advertised to be the solution to the ever-increasing complexity in software development. These techniques offer an easy way to domain-specific abstraction and to a high degree of automation in the coding process. Abstraction and automation lead to higher productivity, easier extensibility and better quality of the software.

Although there are success stories about MDA, MDSD and MDE, the adoption of such techniques in industry is not yet very widespread. (Atkinson, C., Kühne, T., 2003) see the reasons in a still incomplete and not yet fully understood theoretical foundation of MDE. Other research such as (Selic, B., 2008) and (CHAMDE, 2008) investigated this issue from a more practical point of view and identified mainly two kinds of reasons. On the one hand we find so called technical reasons like bad tool support, missing tool documentation, insufficient interoperability between tools, lack of user-friendliness, and others. On the other hand, a lot of programmers simply feel comfortable with their current proven methods of software development. They often only see the discomfort, the difficulty, the threats and dangers but not the benefits in new technology. This lack of awareness, education, and training is often referred to as cultural problems.

Although not all of the tool requirements from (Kent, S., 2002) are fully achieved, there are tool chains such as EMF, GMF and oaw (Eclipse Project) that provide most of the needed functions for MDE at least for smaller-scale projects. In fact, (Thörn, C., Gustafsson, T., 2008) find in a survey among several SMEs that the importance of tool support is "surprisingly low" when it comes to suggest improvements to current practices, whereas "methodology", "increased awareness" and "training" are all mentioned significantly more often.

We are convinced, that the most important obstacle to the adoption of MDE is the appearing discouraging difficulty of meta-modeling, that is due to the lack of methods about how to address the specification of a meta-model or domain specific language (DSL) at the center of each model-driven approach.

In fact, there are papers that present special meta-models or domain specific languages (references in van Deursen, A. et al., 2000). Besides, (Luoma, J. et al., 2004), (Mernik, M. et al., 2005), (van Deursen, A. et al., 2000) identify several high level possibilities to define a meta-model or a DSL. Unfortunately, it remains unclear how to effectively bridge the gap between the domain analysis and the explicit definition of the meta-model or DSL.

In our approach, we build a first version of a meta-model from a traditional object-oriented domain model. As experienced object-oriented software developers should feel comfortable building domain models following for instance the principles of Domain Driven Design (DDD)(Evans, E., 2004), meta-modeling should become easier for them.

Schlechter A., Simon G. and Feltz F. (2009).
FROM AN ABSTRACT OBJECT-ORIENTED DOMAIN MODEL TO A META-MODEL FOR THE DOMAIN - Model Driven Development of a Manufacturing Execution System.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 317-320
DOI: 10.5220/0002252903170320
Copyright © SciTePress

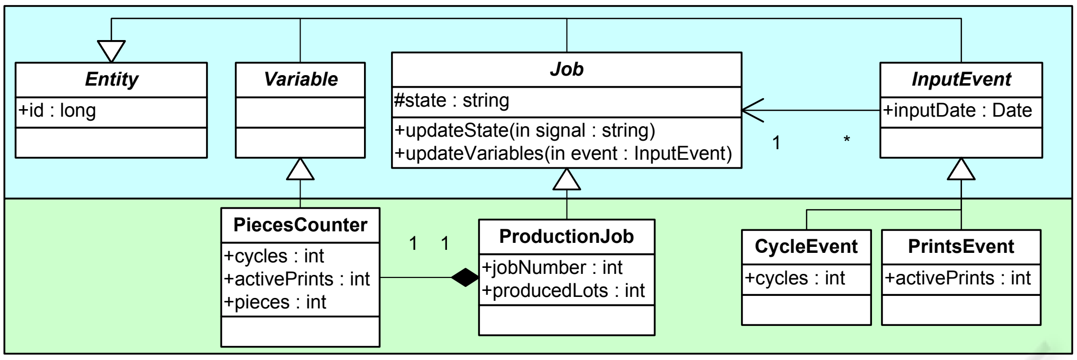

Figure 1: UML Class diagram with an extract of the abstract domain model (upper part) and some concrete example subclasses for a simple MES (lower part).

We will present and illustrate our method using the development of a Manufacturing Execution System (MES) as an example. First of all we introduce the “ubiquitous language” (DDD) for MES, its representation as an object-oriented domain model and the software system architecture (section 2). Based on the architectural description we outline a model-driven approach and explain its benefits (section 3). The meta-model at the center of the approach will be defined based on the object-oriented domain model (section 4).

2 OBJECT-ORIENTED DOMAIN MODEL

Manufacturing Execution Systems deal with the collection, evaluation, analysis, interpretation and visualization of data from production in order to better control the production processes. The central objects of interest in MES are *jobs*. A job produces a *product*. Products have *resource requirements* used to determine what *resources* should be assigned to a job for the production of a given product. Jobs use time on resources, have an internal *state* and may contain several sets of values, so called *variables*, to represent data from quality control and process monitoring for instance. *Input events* may change the state and the variables of jobs. The history of a job’s state is stored in a series of *slots*. Of course, all these objects may have *attributes*. Besides these domain specific objects, there are simple persistent data *entities*.

For the sake of simplicity, we will not go into detail for all of these aspects. An extract from the abstract domain model for MES containing only entities, jobs, variables and input events is depicted at the top of Figure 1. For the implementation of a concrete MES, these abstract classes have to be specialized.

As an example, we assume that there is only one kind of job called ProductionJob that contains exactly one variable PiecesCounter used to count the produced pieces. In order to calculate the produced pieces, we need to know the number of currently active prints in a mold (=pieces produced per cycle). Additionally, we need input events to change the number of active prints (PrintsEvent) and to enter a number of cycles (CycleEvent). The respective classes are represented at the bottom of Figure 1.

As manufacturing execution systems are typically installed between already existing IT-Systems at the customer’s factory, the attributes of these concrete subclasses should be defined to be compatible with the data from the existing systems. In order to keep the implementation of the interfaces between the MES and the surrounding software as simple as possible, we propose a layered architecture with a *service layer* as an outer layer. The services coordinate the access to domain objects in persistent storage via a *data mapper* with the necessary calls to services and object methods from the *domain model* (Patterns from Fowler, 2003).

The service layer provides services for the creation and retrieval of jobs and input events respectively (Figure 2). The definitions of these services contain quite a lot of redundancy when compared to the class diagram in Figure 1. They follow very strict and simple patterns depending on the abstract super-class, the name and attributes of the classes and possible relations between classes. At every modification we need to keep them consistent with the data structures, which makes their implementation and maintenance a time-consuming, repetitive and error-prone task. In fact, most of these services can completely be generated with a little more information than provided by a standard UML class diagram.

```
ProductionJob createProductionJob(int jobNumber, int producedLots)
ProductionJob getProductionJobByJobNumber(int jobNumber)
Collection<ProductionJob> getProductionJobsByState(String state)
void createPrintsEventForProductionJobByJobNumber(int jobNumber, int prints)
Collection<PrintsEvent> getPrintsEventsByProductionJobJobNumber(int jobNumber, Interval p)
```

Figure 2: Some services offered by the JobManager and the InputEventManager in the service layer.

3 DOMAIN SPECIFIC MODEL

Universal modeling languages like the UML are well suited to describe a given software solution. Often, the models are simply abstract representations of the code of an application, thus usually leaving out functional details or spreading these details over a lot of different types of models. Domain specific modeling languages are designed to capture the essence of a domain and the respective models are abstractions of the real world problem to be solved. From such models, we can generate all the abstract models of a solution. In addition, it is often possible to generate some functional details or even a complete application. For illustration, we will now present a possible domain specific model for our example MES-System.

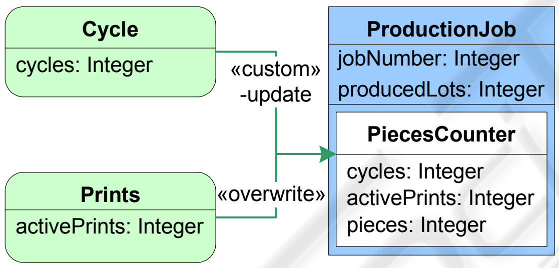

Figure 3: Extract from an MES model corresponding to the object model in the lower part of Figure 1.

In order to distinguish between the different class hierarchies in Figure 1, we model the subclasses using different representations. In Figure subclasses of the InputEvent class are drawn as green rounded rectangles, and subclasses of the Job class are modeled as blue rectangles containing a white rectangle for each owned subclass of the Variable class. The attributes of the classes are modeled just like in UML within the class representing shapes.

Additionally, the new model contains relations between input events and variables that indicate if and how an event updates a variable. In our example, the Cycle event will update the PiecesCounter variable in a custom way that is manually specified in the generated code. The Prints event will overwrite the activePrints attribute of the PiecesCounter variable with the value of its attribute.

It should be clear, that the domain model in the lower part of Figure 1 and the services in Figure 2 can be generated consistently from this new model. Besides, the update relations can be used to generate the major part of the updateVariables() method of the ProductionJob class together with the needed supporting methods within the different input events.

Moreover, this new model contains less technical and more domain specific details. Due to its intuitive meaning, it is easier to read and understand for domain experts.

4 FROM A DOMAIN MODEL TO A META-MODEL

After the presentation of a possible domain specific model and its advantages, we need to formally define the corresponding meta-model.

As a first step, we basically just take the classes of our abstract object oriented domain model in the upper part of Figure 1 and put them as meta-classes in the meta-model in Figure. This leaves us with the meta-classes JobClass, VariableClass, InputEventClass and EntityClass in our meta-model. Instances of these meta-classes represent subclasses of the domain classes Job, Variable, InputEvent and Entity. For instance, the blue rectangle representing the ProductionJob in Figure is an instance of the meta-class JobClass.

All these instances have a name (e.g. "ProductionJob") and may have attributes (e.g. jobNumber). To capture these commonalities, we introduce the super-(meta)class AbstractEntityClass. It has a string-type attribute *name* to take the names and may be associated to several PropertyClasses that represent the attributes. For instance, ProductionJob is an instance of JobClass, that is an AbstractEntityClass with *name*="ProductionJob". Its attribute *jobNumber* with type "Integer" is an instance of PropertyClass with *name*="jobNumber" and *type*=PropertyType::-Integer.

The additional attributes *unique* and *searchable* of PropertyClass indicate whether an attribute of a

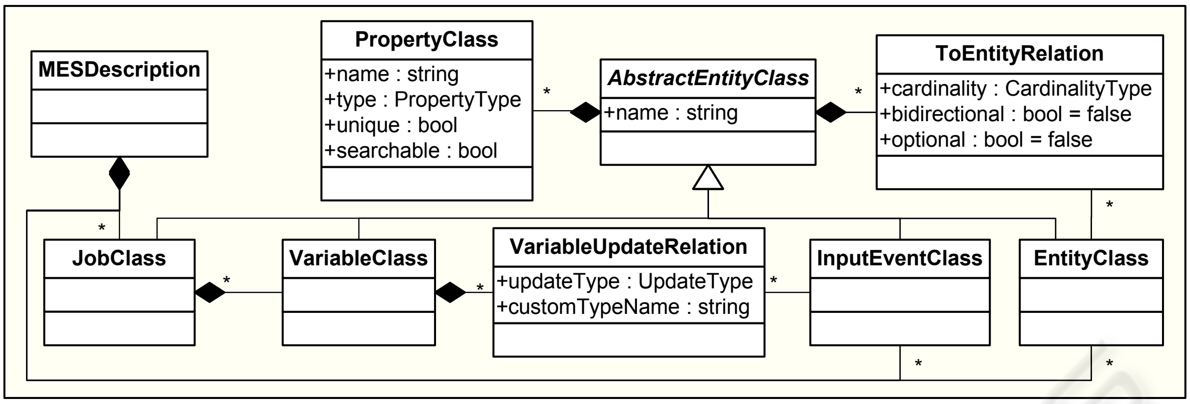

Figure 4: Extract from the MES meta-model.

domain object can be used as an identifier for this domain object and whether it can be used to lookup and retrieve instances of this domain object. They are used to control which accessing services are generated. The service getProductionJobByJob-Number() (Figure 2) for instance is generated because the attribute jobNumber of ProductionJob is marked to be unique and searchable.

Finally, we would like to associate simple data entities to our model elements in order to build normalized data structures. Each such association in a domain specific model is an instance of the ToEntityRelation meta-class. The attributes of this meta-class are used to control the cardinalities and other options of the associations.

As models need a single point of entry for further treatment, all top-level elements of a model are owned by a model-object of type MESDescription.

The only substantial addition to the meta-model when compared to the abstract domain model is the VariableUpdateRelation meta-class used to model which input event updates which variable and how.

5 CONCLUSIONS

We presented a method to design a meta-model starting from an abstract object-oriented domain model. Models conforming to this meta-model can be used to generate a concrete specialization of the domain model together with supporting code, where all elements are guaranteed to be consistent with one another. Several important goals of model driven software development such as higher productivity, easier extensibility and better quality can be achieved using this simple method. The method can be easily applied by is easily understandable by facilitates model driven engineering for experienced object-oriented developers.

REFERENCES

Atkinson, C., Kühne, T., 2003. Model-driven development: a metamodeling foundation. In IEEE Software. Vol. 20. No. 5. Sept-Oct 2003.

CHAMDE, 2008: 1st Int. Workshop on Challenges in Model Driven Software Engineering. Sept 28th. Toulouse. France. (see Posse, Rutle, and Mohagheghi)

Eclipse, 2009: Eclipse Modeling Project EMF,GMF, oaw: http://www.eclipse.org/modeling/ (2nd april 2009) http://www.openarchitectureware.org/ (2nd april 2009)

Evans, E., 2004. *Domain-Driven Design Tackling Complexity in the Heart of Software*. Boston. Addison-Wesley.

Fowler, M, 2003. *Patterns of Enterprise Application Architecture*. Boston. Addison Wesley.

Kent, S., 2002. *Model Driven Engineering*. In Proceedings of the Third International Conference on Integrated Formal Methods. LNCS Vol. 2335, 2002.

Luoma, J., Kelly, S., Tolvanen, J-P., 2004. *Defining Domain-Specific Modeling Languages: Collected Experiences*. In Proc. of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04), Technical Reports, TR-33, University of Jyväskylä, Finland 2004.

Mernik, M., Heering, J., Sloane, A. M., 2005. *When and How to Develop Domain-Specific Languages*. In ACM Computing Surveys. Vol. 37. Issue 4. Dec 2005.

Selic, B., 2008. *Personal Reflections on Automation, Programming Culture, and Model-based Software Engineering*. In Automated Software Engineering. Vol. 15. Issue 3-4. Dec 2008.

Thörn, C., Gustafsson, T., 2008. *Uptake of Modeling Practices in SMEs Initial Results from an Industrial Survey*. In 2008 Int. Workshop on Models in Software Engineering. Leipzig. Germany. 2008.

van Deursen, A., Klingt, P., Visser, J., 2000. *Domain Specific Languages*. In ACM SIGPLAN Notices. Vol. 35. Issue 6. June 2000.