Integration of Object Oriented Domain Modeling and Meta-Modeling

Antoine Schlechter, Guy Simon, Fernand Feltz

Département Informatique, Systèmes et Collaboration (ISC)
Centre de Recherche Public Gabriel Lippmann
41, rue du Brill
L-4422 Belvaux
schlecht@lippmann.lu
simon@lippmann.lu
feltz@lippmann.lu

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. One of the major reasons is the appearing discouraging difficulty of meta-modeling. This paper illustrates the relations, dependencies and differences between a traditional abstract object-oriented domain model and a meta-model for the same domain. It presents a new approach to model driven engineering of enterprise application software that integrates domain modeling and meta-modeling in order to take full advantage of both traditional and generative software development methods.

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. [AK03], [Kü06] and [He06] see the reasons in a still incomplete and not yet fully understood theoretical foundation of MDE. Other researchers such as [PD08], [Sel08], [RR08], and [MFM08] 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 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 shortcoming of awareness, education, and training is often referred to as cultural problems.

Although not all of the tool requirements from [Ke02] are fully achieved, there are tool chains such as EMF, GMF and oaw [Ecl09] that provide most of the needed functions for MDE at least for smaller-scale projects. In fact, [TG08] finds 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 [KK03], (references in [DKV00]). Besides, [LKT04], [MHS05], and [DKV00] 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.

[RJ96] proposes domain specific languages on top of a framework in order to make this framework more comfortable to use. [AC06] analyses the possibilities to design Framework Specific Modeling Languages (FSML) with roundtrip engineering. [Sa07] introduces annotations to the implementation of a framework that may be used to generate a DSL for this framework. These DSLs remain very close to the technical details of a given framework. They are defined on top of existing finished frameworks.

Since we would like to take the benefits of model driven engineering during the whole development process, we present an approach that integrates long-established object-oriented domain modeling and meta-modeling. First we define a traditional abstract object-oriented domain model. The idea is to generate the concrete subclasses with their supporting code from a so-called domain specific model. In order to formally describe such a model, we need to define a meta-model. A first version of this meta-model can be derived from the abstract domain model. After that, we may choose for each more advanced feature of the system whether to include it in the object-oriented domain model or in the meta-model. As experienced object-oriented software developers should feel comfortable building domain models following for instance the principles of Domain Driven Design (DDD)[Ev04], meta-modeling, the first, most important, most difficult and most discouraging step in model driven software development, should become easier for them just by following the concrete guidance from our method.

To present and illustrate our method, we will use the development of a Manufacturing Execution System (MES) as an example. First of all we introduce the "ubiquitous language" (from 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). Both the domain-model and meta-model are an integral part of the proposed MDE approach and as such, they may be extended separately to take full advantage of both traditional and generative software development methods (section 5). The proposed method has been use to develop a concrete manufacturing execution system for a plastics injection molding company (section 6).

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 must 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. A first extract from the abstract domain model for MES containing only entities, jobs, variables and input events is depicted at the top of Figure 1.

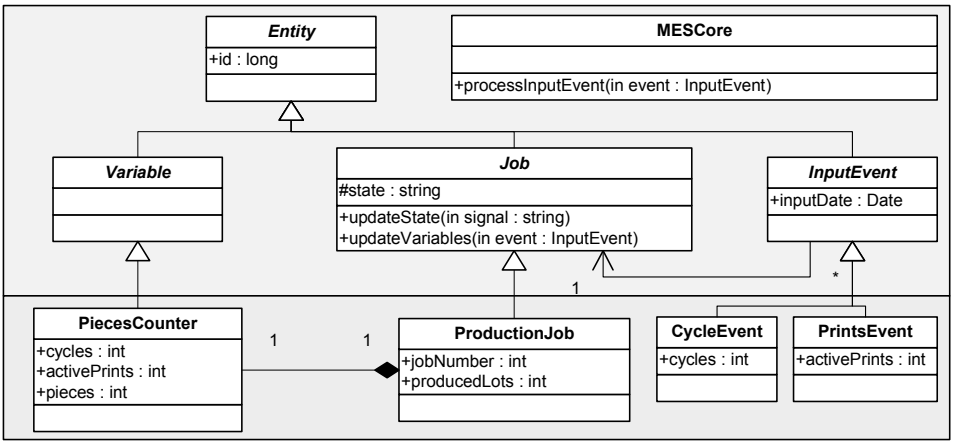

Figure 1: Class diagram with an extract of the abstract framework classes (upper part) and some concrete example subclasses for a simple MES (lower part).

This UML model represents the static abstract class structure of an MES implementation. It contains the data structure and some very basic operations and services. The processInputEvent() service of the MESCore class coordinates the different operations

on a job, its state, variables and history involved in the processing of an input event. 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.

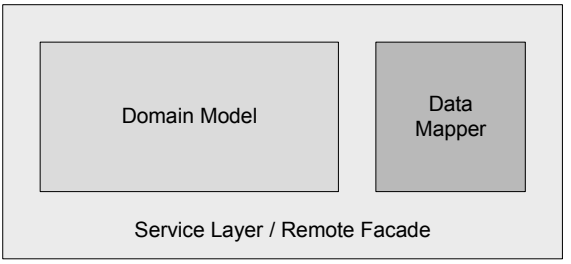

Figure 2: Architecture of the MES-Backend with a 'Domain Model' and a 'Data Mapper' on the inner layer and a 'Service Layer' implemented as a 'Remote Façade' on the outer layer. (Patterns from [Fo03])

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.

```
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)
Collection<PrintsEvent>
         getPrintsEventsByProductionJobJobNumber(int jobNumber, Interval p)
```

Figure 3: Some services offered by the JobManager and the InputEventManager in the remote façade / service layer.

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 / remote façade that exposes useful functions to remote and local clients as an outer layer. The services in the remote façade / service layer 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 (Figure 2). For a detailed explanation of the patterns remote façade, service layer, data mapper and domain model, we recommend [Fo03].

The service layer provides a JobManager and an InputEventManager with services for the creation and retrieval of jobs and input events respectively. Additionally, the creation

services for input events also process the created events by calling the MES-Core processInputEvent() service. These managers offer among others the services listed in Figure 3.

This short list of services contains quite a lot of redundancy when compared to the class diagram in Figure 1. Most of these services 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. Thus, the implementation and maintenance of these services is a time-consuming, very 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.

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.

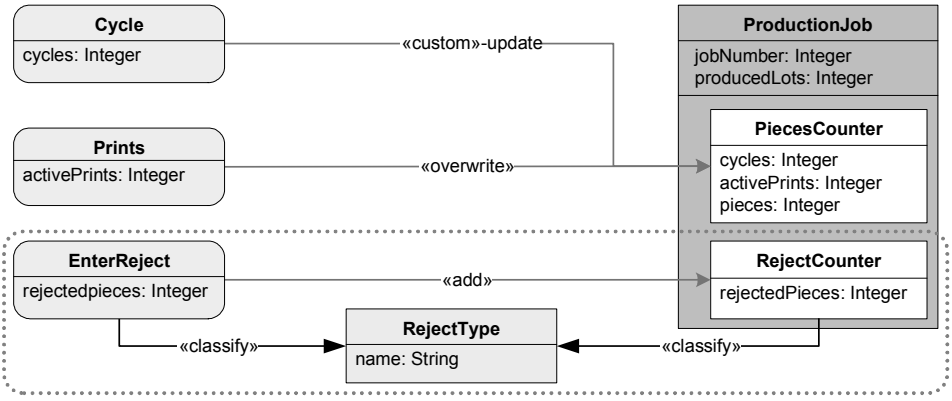

Figure 4: Extract from an MES model. The non-dotted area corresponds to the object model in the lower part of Figure 1; the dotted area represents the object model in Figure 5.

As seen in the analysis of the services needed in the remote façade, we need a distinction between the different class hierarchies in Figure 1. Thus, we model the subclasses using different representations. In Figure 4 subclasses of the InputEvent class are drawn as green rounded rectangles (left-hand side), direct subclasses of the Entity class are represented by gray rectangles (bottom), and subclasses of the Job class are modeled as blue

rectangles (right-hand side) 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.

In addition to Figure 1, we added relationships between input events and variables that indicate if at all and how a given event updates a variable. In our example, the Cycle event will update the PiecesCounter variable in a custom way to be further specified manually in the generated code. The Prints event will overwrite the activePrints attribute of the PiecesCounter variable with the value of its attribute. Within the dotted area of Figure 4, we added an EnterReject event and a RejectCounter variable that both use RejectType to classify their attribute rejectedPieces. The corresponding data structures are given in Figure 5. In fact, the RejectCounter variable will have a hashtable to map reject types to the corresponding total number of rejected pieces.

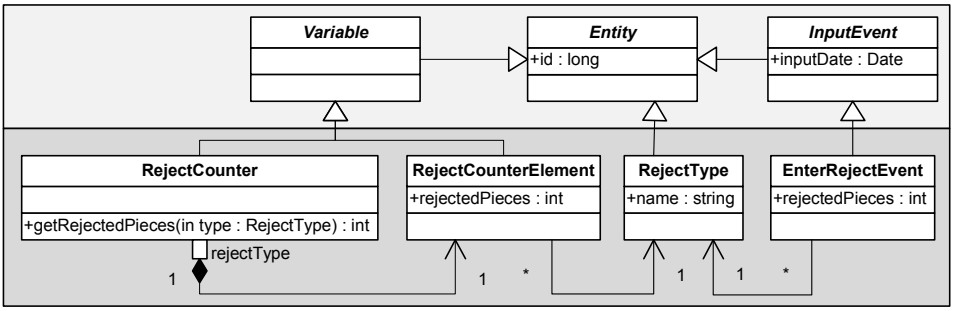

Figure 5: Class diagram corresponding to the dotted area in the MES model of Figure 4.

For now, Figure 4 is just a picture and we need to define its formal semantics in order to make it a model [Kl07]. We have drawn this new model with clear "translational semantics" in mind. In fact, we constructed its elements directly from the subclasses of our abstract domain model they represent. It should be clear, that the models in Figure 1 and Figure 5 as well as the services in Figure 3 can be generated from the model in Figure 4.

In this new model, we left out technical details and added domain specific details like the update relationships between input events and variables. This additional information 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. Besides, the new model is more readable and has an intuitive meaning that is easily understandable by domain experts.

4 From Domain Modeling to Meta-Modeling

After the presentation of a possible domain specific model and its advantages, we need to formally define the corresponding meta-model. Despite the higher effort for tool building involved with a real new meta-model, we chose this way because of its other advantages over an UML profile or an UML extension [WS07].

As a first step in the definition of the meta-model, 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 6. After this first step, we find 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 4 is an instance of the meta-class JobClass.

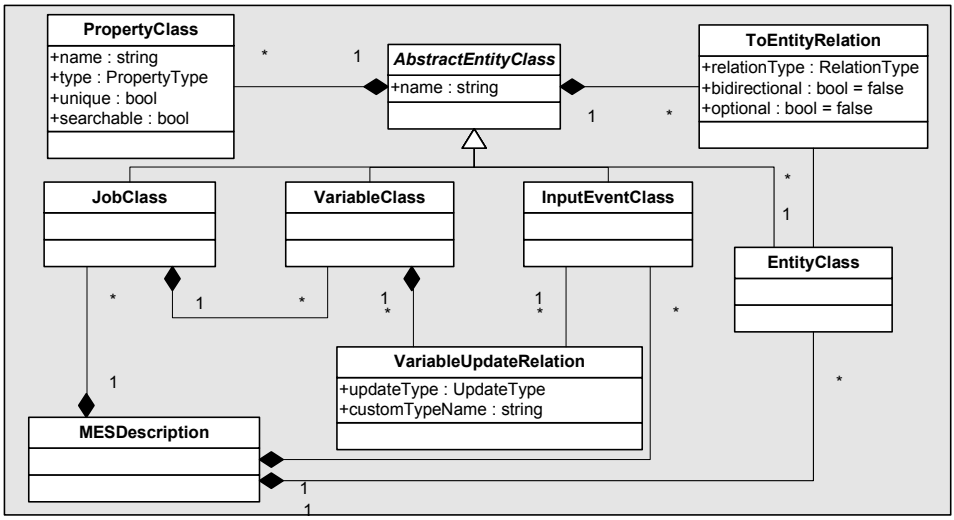

Figure 6: Extract from the MES meta-model.

All these instances should have a name (“ProductionJob”, “RejectCounter”, “PrintsEvent”, “RejectType”, …). It should be possible, to define attributes for these instances (jobNumber, rejectedPieces, …). To capture these commonalities in a central element, we introduce the common super-(meta-)class AbstractEntityClass in our meta-model. It has a string-type attribute name to take the names of the instances. AbstractEntityClasses may have several PropertyClasses to represent the attributes of their instances. 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 are used to indicate whether an attribute of a domain object (=instance of PropertyClass) 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 should be generated. The service getProductionJobByJobNumber() (Figure 3) for instance is generated because the attribute jobNumber of ProductionJob is marked to be unique and searchable.

Besides attributes (instances of PropertyClass), we would like to associate simple data entities to our model elements (instances of JobClass, …) 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 type (classifying or not, cardinalities) and other options of the associations.

Variations of this AbstractEntityClass-pattern will be a part of practically every meta-model that is used to model traditional data structures. Another returning aspect is the need for models to have one single point of entry. Therefore all top-level elements in a model (those that are not owned by other model elements) must be owned by one object of a special type. In our case this is MESDescription.

The only substantial additions to the meta-model are the composition relationship between JobClass and VariableClass indicating that jobs may contain variables and the VariableUpdateRelation meta-class to model which input events update which variables and how.

5 Integrated Domain Modeling and Meta-Modeling

Our approach contains two separate development processes: the first one is about the development of applications; the second one is about the development of tools that are used in application development (Figure 7).

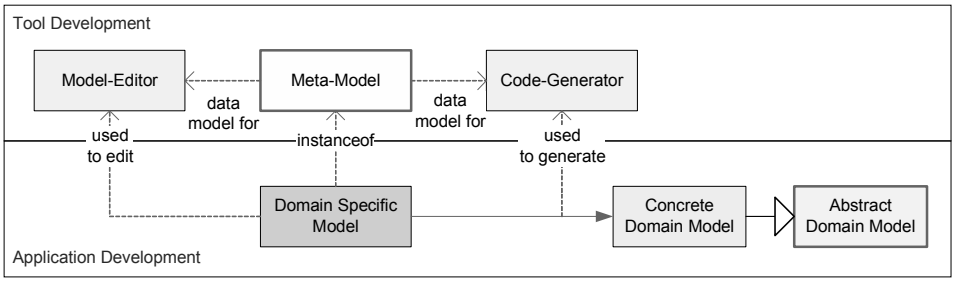

Figure 7: Overview of the integrated domain modeling and meta-modeling approach.

We start the application development traditionally with an abstract object-oriented domain model. Instead of manually extending and specializing this abstract domain model into a concrete one, we build domain specific tools to do this more efficiently. At the beginning of the tool development, we lift the classes from the abstract domain model into a meta-model. This meta-model can be seen as a data-model to store and interchange domain specific models [Sei03]. It is used as data-model for a model-editor and a code-generator. The model editor is used to create and edit domain specific models that are transformed into a concrete domain model by the code generator. Both these tools could be developed following traditional software development methods. However, there exist frameworks like GMF and oaw [Ecl09] that facilitate this development significantly.

This being said, we have two separate places where we can model the different aspects of a software system: the abstract domain model and the meta-model. For every aspect

we may decide to either model it in the abstract domain model with a manual implementation possibly completed by an extension of the code generator, or to model it in the meta-model with its implementation becoming an integral part of the code generator. The most significant difference between domain modeling and meta-modeling in our approach is changeability. An aspect modeled in the meta-model will usually be transformed into static code. Thus, a change here will have us to deploy a new version of the software. Modeling the same aspect in the abstract domain model usually allows us to make changes at runtime. To get there, modeling of behavior at this level will force us to implement complex interpreters.

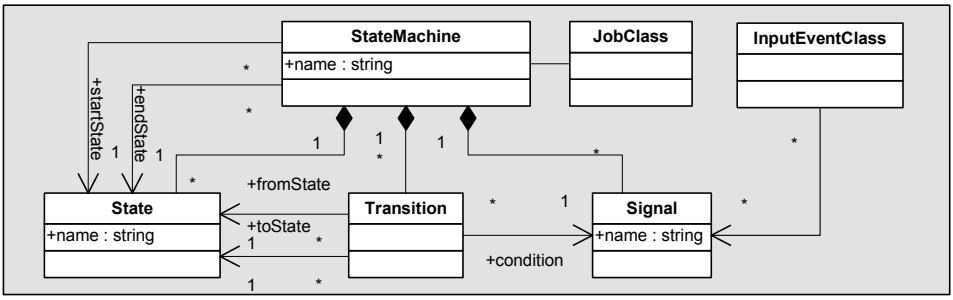

Figure 8: Extension of the meta-model to allow the modeling of state machines for jobs.

As an example for an extension of the meta-model, we present the modeling of the state of a job. A job can have different states at different points in time. Input events may cause jobs to change their state. The possible states of a job and the transition between them with the respective conditions form a state machine. In a production system, we may not change the state machine for a type of job as this will falsify the history of past jobs. Change can only be accomplished by adding a new type of job with a new state machine. Hence we model the state machines in the meta-model and generate static code for them. Figure 8 contains the corresponding meta-model extension and Figure 9 contains the domain specific model with the state machine for ProductionJobs.

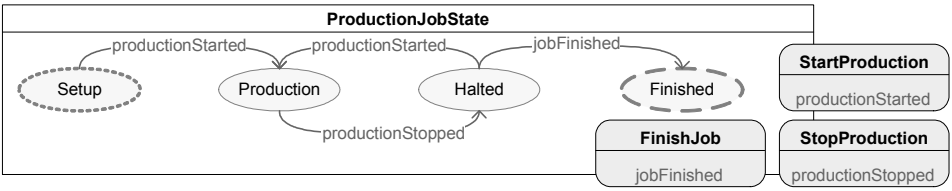

Figure 9: Model of the different states for ProductionJobs.

On the other hand, the history of a job as a series of slots is modeled in the domain model (Figure 10). Each Slot stores the state of its job in the period of time before its startEvent and its endEvent. Besides this data structure for slots, we need a structure to make some statistics over the usage of time for a given job. The abstract class BaseTimeDistibution allows to sum up the total time of the slots that are given to it. In our example we are interested in how long a job spend in its different states. Therefore we generate the ProductionJobsDistribution for ProductionJobs that contains distribu-

tions for each one of the possible job states. When adding slots to this ProductionJobsDistribution, the slots are recursively added to the distributions corresponding to the job's state stored in the slot. As all information needed to generate these TimeDistributions is already contained in our model, there is no need to extend the meta-model.

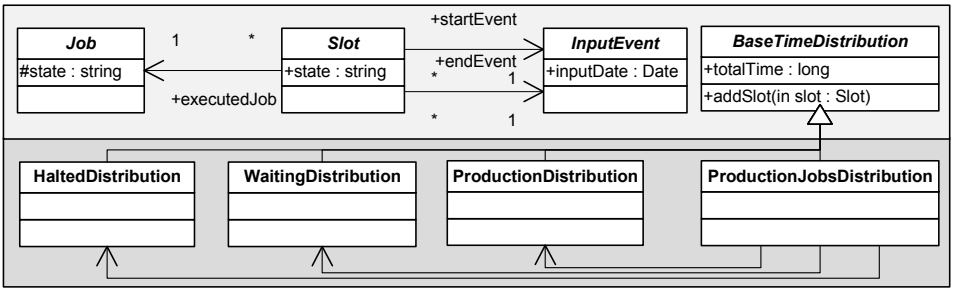

Figure 10: Extension of the framework-model to build statistics on the time a job spent in different states.

If we had modeled the VariableUpdateRelation or the state machine in the domain model, we would have needed to implement interpreters to execute the stored update relations or state transitions.

6 Experimental Results

We implemented the whole MDE framework for MES, including meta-model, model-editor and code-generator using Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) and openarchitectureware (oaw) [Ecl09]. As target platform we chose Java 5 with Enterprise Java Beans (EJB) 3.0. The structure of the code generator follows the principles of the Model Driven Architecture (MDA) [OMG03], i.e. it consists of several modules responsible for different architectural elements (CIM to PIM) and each module generates the needed Java code (PIM to PSM to code).

For the implementation of the GUI we used QT-Jambi and for the generation of reports we used Crystal Reports. Both tools fit well in our model driven approach with their good WYSIWYG designers.

The generated system has been tested in a real world environment using the JBoss application server at an injection molding company. During the testing phase, several bugs have been identified that could easily be fixed in the framework and code-generator. Some feature requests arose that could be implemented quickly by small additions to the meta-model and code-generator. The most difficult parts have been the GUI-enhancements to make the new features visible and editable.

The implementation of the backend of our Manufacturing Execution System (Figure 2) contains 1541 lines of code in 45 classes, the complete MES meta-model contains 22 meta-classes with 151 model elements (=classes, attributes, and associations) and the code-generator templates contain 4356 lines of code. The complete model of the manu-

facturing execution system for the injection molding company is depicted in Figure 11. It contains 175 model elements. From these we generate 90 classes with a total of 7525 lines of code that are recreated every time the generator is run. Another 37 subclasses with 594 lines of code are created once at the first generator run. These may be manually modified to implement some custom functions. In our system, we added a total of 128 lines of code to 21 of these classes. This gives us a fix cost of 151 model elements plus 5897 lines of code and a variable cost of 175 model elements plus 128 lines of code for this one system.

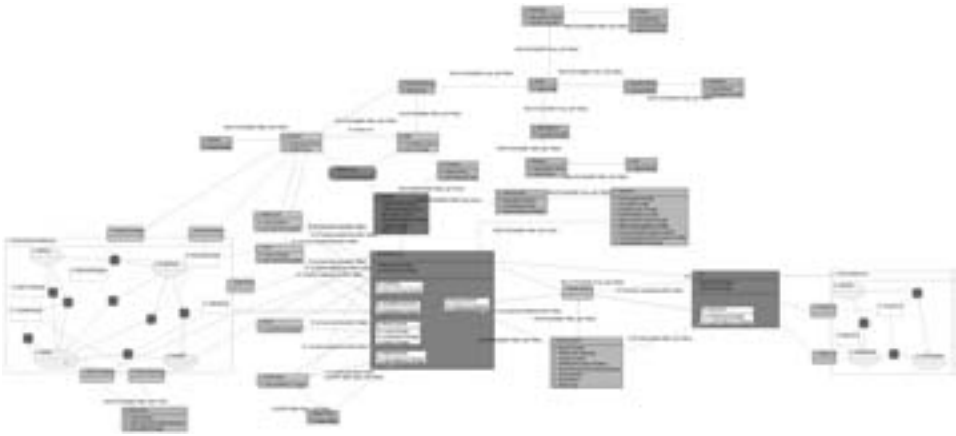

Figure 11: Complete Model of the implemented MES-System.

The overall system architecture would not be different if we had not chosen to use code generation. Thus, we need to compare the effort to write the 8119 generated lines of code with the effort to write the 4356 lines of code in the code generator and the 326 model elements. Around half the lines of code in the code generator are final code templates. This means, that each code template is reproduced around four times in average. In consequence, only around a quarter of the generated code has to be tested to be sure that "all" code works correctly whereas the manual code has to be tested completely. Additionally a change in one implementation pattern due to an error affects around four code blocks that need to be changed independently but consistently in the manual code. The renaming of a model element in compliance with all naming conventions might propagate to even more different places. Even if manually writing these lines seems attractive, testing them appropriately and keeping them consistent over time will be very hard.

Besides the slight numerical gain in productivity in the development of this first system, the MDE approach will speed up the development of further systems due to the small variable cost per system. Additionally there is a huge increase in maintainability due to the automatically assured consistence between all code parts involved in the handling of the same objects. The possibility to systematically add new functionality either in the manually implemented framework (domain modeling) or in the code generator (meta-modeling) makes the systems easily extensible. Finally, the validation of the code gen-

erator templates by testing a "minimal" system guarantees the quality of a complete system.

7 Conclusion

We presented an approach to model driven engineering, where the central meta-model is defined starting from a traditional abstract object-oriented domain model. Using the differences between modeling at the abstract domain level and modeling at the meta-level, we may integrate both levels and choose for each function of a software system the optimal way of modeling and implementation.

The models conforming to a meta-model defined following our method can be used to generate a concrete specialization of the underlying abstract domain model where all elements have an assured quality and are guaranteed to be consistent with one another. Thus, our method achieves the most important goals of model driven software development such as higher productivity, easier extensibility and better quality.

As meta-modeling is the most difficult and discouraging step in model driven engineering, the concrete explanations in our method about how to start the definition of a meta-model, what to put into the meta-model and what to keep in the abstract domain model, make the access to MDE easier for experienced object-oriented developers.

References

[AK03] Atkinson, C., Kühne, T., 2003. Model-driven development: a metamodeling foundation. In IEEE Software. Vol. 20. No. 5. pp 36-41. September-October 2003.

[AC06] Antkiewicz, M., Czarnecki, K., 2006. Framework-Specific Modeling Languages with Round-Trip Engineering. In LNCS Vol. 4199. Springer. Berlin / Heidelberg. 2006.

[Ecl09] Eclipse Modeling Project EMF, GMF, oaw: http://www.eclipse.org/modeling/ http://www.openarchitectureware.org/ (accessed 18th may 2009)

[Ev04] Evans, E., 2004. Domain-Driven Design Tackling Complexity in the Heart of Software. Boston. Addison-Wesley.

[Fo03] Fowler, M, 2003. Patterns of Enterprise Application Architecture. Boston. Addison Wesley.

[He06] Hesse, W., 2006. More matters on (meta-)modeling: remarks on Thomas Kühne's "matters". In Journal on Software and Systems Modeling, Vol. 5, No. 4. pp. 387-394. December 2006.

[Ke02] Kent, S., 2002. Model Driven Engineering. In Proceedings of the Third International Conference on Integrated Formal Methods. LNCS Vol. 2335, pp 286 – 298. 2002.

[Kl07] Kleppe, A., 2007. A Language Description is More than a Metamodel. In 4th Int. Workshop on Software Language Engineering. Nashville. USA. October 2007.

[KK03] Koch, N., Kraus, A., 2003. Towards a Common Meta-model for the Development of Web Applications. In LNCS Vol. 2722. Springer. Berlin. 2003

[Kü06] Kühne, T., 2006. Matters of (Meta-)Modeling. In Journal on Software and Systems Modeling, Vol. 5, No. 4. pp. 369-385. December 2006.

[LKT04] 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

[MHS05] Mernik, M., Heering, J., Sloane, A. M., 2005. When and How to Develop Domain-Specific Languages. In ACM Computing Surveys (CSUR). Volume 37. Issue 4. pp. 316-344. December 2005.

[MFM08] Mohagheghi, P., Fernandez, M. A., Martell, J. A., Fritzsche, M., Giliani, W., 2008. MDE Adoption in Industry: Challenges and Success Criteria. In 1st Int. Workshop on Challenges in Model Driven Software Engineering. September 28th. Toulouse. France.

[OMG03] Object Management Group (OMG), 2003. MDA Guide Version 1.0.1, OMG Document/03-06.-01. 2003.

[PD08] Posse, E., Dingel, J., 2008: A Foundation for MDE. In 1st Int. Workshop on Challenges in Model Driven Software Engineering. September 28th. Toulouse. France.

[RJ96] Roberts, D., Johnson, R., 1996. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In 3rd Conference on Pattern Languages and Programming. Addison-Wesley. 1996.

[RR08] Rutle, A., Rossini, A., 2008. A Tentative Analysis of the Factors Affecting the Industrial Adoption of MDE. In 1st Int. Workshop on Challenges in Model Driven Software Engineering. September 28th. Toulouse. France.

[Sa07] Santos, A. L., 2007. Automatic Support for Model-Driven Specialization of Object-Oriented Frameworks. In 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. pp. 923-924. Montreal. Quebec. Canada. 2007

[Sei03] Seidewitz, E., 2003. What Models Mean. In IEEE Software. Vol. 20. No. 5, pp 26 – 32. September 2003.

[Sel08] Selic, B., 2008. Personal Reflections on Automation, Programming Culture, and Model-based Software Engineering. In Automated Software Engineering. Vol. 15. Issue 3-4. pp 379 – 391. December 2008.

[TG08] 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.

[DKV00] van Deursen, A., Klingt, P., Visser, J., 2000. Domain Specific Languages. In ACM SIGPLAN Notices. Volume 35. Issue 6. pp. 26-36. June 2000.

[WS07] Weisemöller, I., Schürr, A., 2007. A Comparison of Standard Compliant Ways to Define Domain Specific Languages. In LNCS Vol. 5002, pp. 47-58. Springer. Berlin / Heidelberg 2008.