Model integration in domains requiring user-model interaction and continuous adaptation of meta-model

Peter Krall, 2006-10-01
Cortex Brainware Consulting & Training GmbH, Kirchplatz 5, D-82049 Pullach, Germany

Abstract

Possible architectures for meta-models for domain specific model driven development are compared in the context of a domain – exploration engines for patterns in the dynamics of financial markets – that require interaction between domain expert and model, continuous development of the meta-model and yield the necessity to provide the domain expert with means to express rather abstract mathematical concepts. The focus will be on the decisions for structural integration versus dynamic integration, and for integration within an object oriented meta-model versus integration by mapping between formal grammars. It will be argued that structural integration within an object oriented meta-model is the most promising approach for the particular task.

Motivation

Domain specific modelling is a concept for increasing productivity in software development by integrating the development of models based on domain specific concepts and executable models. Ideally, the domain expert should not need to bother with the development of an executable implementation model from their domain model as the mapping will be predefined in the meta-model. However, in practice it will often be more or less impossible to define the mapping in advance, yielding the question how can the architecture of a meta-model be designed to support model integration and allow for continuous development of the meta-model at the same time. Moreover, it will often be an important task of the model to interact with the user and assist them with the design of the solution, yielding the demand for something like a design-time executable functionality of the model.

Tools for research in the financial markets are examples of these types of situations: Many experts believe, that the dynamics of speculative markets show recognizable patterns that can be exploited by interpreting initial segments of the patterns as triggers for own action [Sw95]. The working hypothesis is that such patterns are statistical side effects of invariance in the momentums of a dynamic system that is constituted by superposition of interactions between many agents and occasional perturbations, such as national bank interventions. Such systems are unlikely to be predictable by a closed theory. Yet they will often expose considerable constraints on the set of possible trajectories through state space that allow negative assertions on trajectories [Ba91]. Although assertion of such constraints is logically negative – the system will *not* do this or that - they may yield exploitable assertions in particular situations – e.g. “If development of some ratio x/y becomes directed after a period of strong fluctuations, then it will not pass a threshold t without a previous outburst into the opposite direction, where t depends on the strength of previous fluctuation by ….” This will say nothing about future development most of the time but may be sufficient to identify profitable trades occasionally, which is enough to justify considerable efforts.

The logical structure of assertions on constraints is that of statements with quantifications on sets of trajectories through state space of a complex system. Such statements – e.g. the one cited above – need to be expanded into sets of empirically testable formula by crawling parameter spaces and generating closed formulas for empirical verification. That motivates

the idea to provide the domain expert with the computational power of exploration engines that generate and test manifolds of hypotheses on market dynamics. Domain expertise is needed to identify interesting hypotheses. Taking into account that the domain model will already be formalized, the question arises, whether this job may be a candidate for automatic model transformation. This yields the idea to use model driven development techniques for providing the domain expert with the facilities for *generic rapid exploration engine development* (greed). Essentially solutions in this domain are models of AI-systems (admittedly rather dull AI), that search for patterns in the dynamics of a complex system.

The key observation concerning requirements for greed is that the development of the domain model is actually the goal. The implementation serves as a vehicle for the domain expert to explore the domain and the executable basically serves as an instrument for development of the domain model. This situation has a number of consequences:

- The objects to be developed in the domain will show 'rich' behavior. The behavior of such machines needs to be definable in much more domain-specific concepts than those of traditional programming languages.
- The designer will often switch between different levels of abstraction, e.g. between definition of a meta-strategy for exploration of trading strategies and specification of parameter sets for instantiation of model-level trading strategies as instances of the meta-level types.
- The process of development will often be interactive in the sense, that the domain expert's ideas for specification of new exploration engines will be inspired by observation of previously designed ones.
- There is no chance to develop the complete domain meta-model and it’s mapping to corresponding executable implementation models in advance, even if this would be desirable. The development strategy thus must support integrated development on domain view-, implementation- and meta-model-level.

On the other hand, aspects like adaptability to different target platforms or interoperability are irrelevant here and no trade-off should be accepted in favor for such aspects.

The needs sketched above constitute the context for considerations concerning an adequate architecture of the meta-model for domain specific modelling in the respective domain. In this paper the focus will be on two aspects of the meta-model:

- Is the meta-model based on structural integration in the sense of a single model principle[Pa02] or does it work with separate domain- and implementation models and explicit transformation?
- Is the integration of views defined within an object-oriented meta-model or on basis of a mapping between formal languages?

To proceed with, the consequences of these two logically independent questions will be sketched out and the structural integration within an object oriented meta-model for projects like greed will be argued.

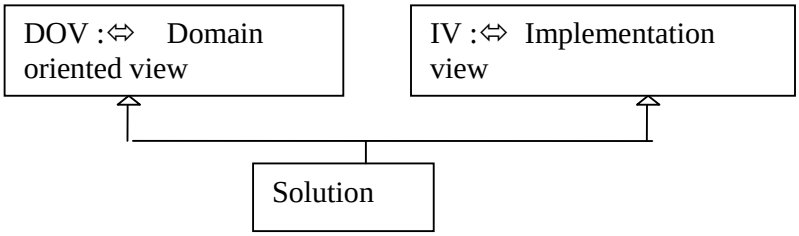
Views

Many scenarios for MDD can be described in terms of integration of design models that specify the solution in domain-specific concepts and platform/language-specific

implementation models [St05]. The transformation is done automatically or semi-automatically by code generators which specialize given domain model for a particular platform by adding information needed for a complete implementation model. MDD concepts therefore often focus on generalization / specialization relations between models. This looks very natural when starting from the idea that model driven development and domain specific modelling serve to increase productivity by providing the designer with an environment that allows them to describe their model using domain-adequate concepts rather than low-level abstractions. [Gr04, St05, To06] The automatic generation of the implementation – or at least a skeleton thereof – can then be seen as a specialization of the design for a particular platform or machine – e.g.: a domain model of functions for a smart phone can be automatically expanded into an implementation model for a particular device through a tool chain that itself can be developed using a meta-CASE tool.

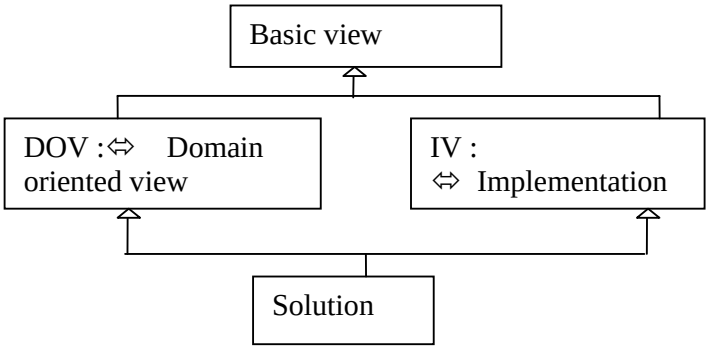
Strictly speaking, it is often a simplification to call the implementation view a specialization of the domain view because the former contains information which is not contained in the latter – from diagram layouts to references to formula, video clips illustrating game-theoretical considerations through visualization of computer simulations, or whatever kind of things which are important for the domain model developer but not executed at runtime. Notwithstanding that this point may be irrelevant for much of the practical work in model driven development, it is important for the meta-level issues of concepts for the integration of development in different views. For this purpose domain and implementation view actually need to be conceived as different generalizations of the underlying complete solution without presupposing that one view were a specialization of the other:

Diagram 1: General scheme of views



Since domain view and implementation are not orthogonal abstractions of the integrated solution, it may make sense to derive both views from a common base:

Diagram 2: General scheme of views with common basic view.



Moreover, there will often be more than two relevant views, not only in the trivial sense of expanding and collapsing regions of code in an IDE, but also in a non-trivial sense. For the

sake of simplicity, the following considerations will nevertheless be formulated on base of the assumption of only two views:

- The domain view: This is the solution as the domain expert sees it, formulated in domain specific concepts.
- The implementation view: This is a description of the solution in a form, which allows transformation into an executable program with means that are considered stable for the purpose of development of solutions within the domain.

The implementation view does not need to be a complete standard-language solution, but may still presume arbitrary complex frameworks, libraries, compilers, code-generators or other tool chain elements. To the extent that these prerequisites are considered as an invariant part of the meta-model, their development nevertheless does not need to be integrated with that of the solution - but is an independent task. It may be noted, though, that confidence in the sufficiency of the tools as they are at the beginning of the project need not be unlimited.

Structural vs. dynamic view integration

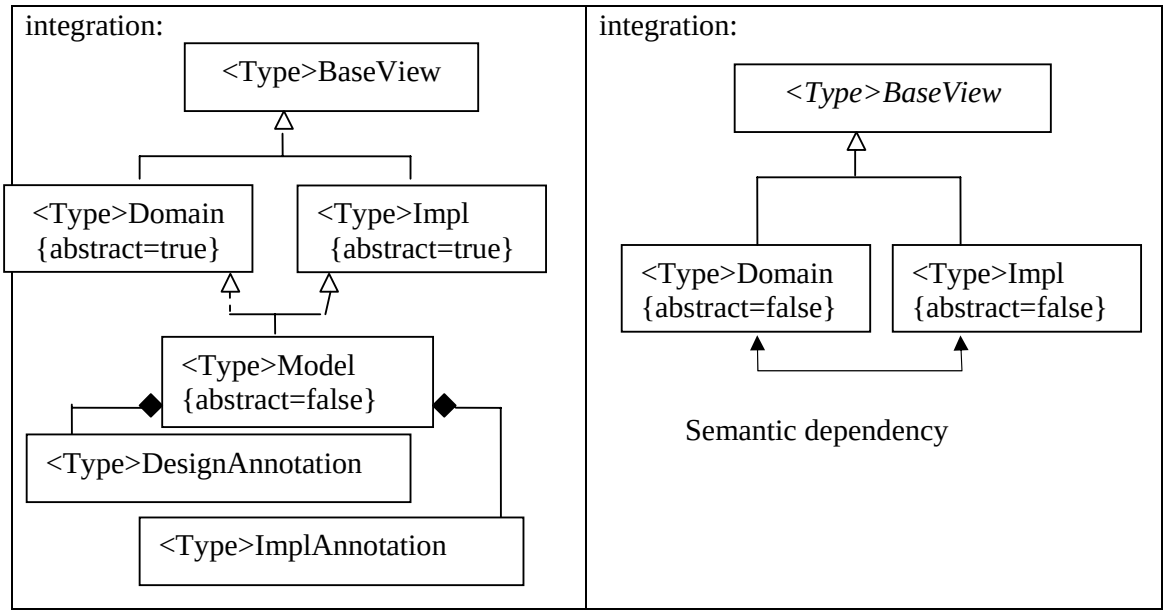
Obviously, a pair of different abstract models must fulfill many constraints in order to have an interpretation as two views on the same solution. Outside model driven development environments it's the task of the software engineer to maintain consistency with little support for preventing the notorious tendency of model views from drifting apart. Model driven development aims at providing mechanisms for integration of development of different views in a way that helps to prevent consistency. One of the basic decisions is, to determine the need to separate the domain and implementation model with a mechanism for propagation of information between the two, or to adopt the single model principle [Pa05].

The single model principle is characterized by the existence of only one model at any time. The different views present different aspects of this unique model. The integration is defined by the structure of the relation between the integrated model and the individual views. This motivates the term 'structural' integration.

Alternatively, different views may be built from model elements with different individual identity. This implies a potential for semantic incoherency. Therefore a mechanism is needed, that is able to transform a pair of incoherent views into a coherent pair of modified views by modification of either one or both views, using either one or both views as input. The possible mechanisms range from one-way code generation to complex bi-directional synchronization mechanisms. In any case there are explicit transformations from one model state to another one. This motivates the term 'dynamic' integration.

The main differences may be summarized as follows:

Structural integration	**Dynamic integration**
Shared properties of model elements in domain and implementation view exist as properties of underlying elements of an integrated model. Views therefore only provide access to differently filtered properties of the same unique model.	Domain view and implementation view actually correspond to two separate models which consist of disjoint sets of model elements with different identity. Correspondence between both views thus means that constraints for values of properties of different objects are fulfilled.
Scheme for the structure of meta-model classes representing a <Type> for structural	Scheme for the structure of meta-model classes representing a <Type> for dynamic



The structural integration scheme for construction of model elements corresponds to a single model, extended by two types of annotations, whereas the scheme for dynamic integration corresponds to construction of two structurally independent models, which need mechanisms for reaching consistency by explicit transformations.

Structural as well as dynamic integration may be combined with either object oriented or grammar oriented representation, as will be demonstrated below.

Dynamic integration allows for isolation of the views' meta-models as such and the transformations between them. The meta-models of different views may be captured in two grammars or two meta-model type systems. Transformations may then be represented by templates or by specialized synchronization methods. There are natural slots for adaptation of mappings between domain- and implementation meta-model through adaptation of templates. Dynamic integration is flexible and may boost productivity [Ko05, To06].

However, the arguments for explicit transformation depend to some extent on the completeness of code generation. If this is 100%, then the arguments are convincing [To06]. In a scenario where the domain meta-model is unlikely to be completely understood in advance, this is probably not realistic. This can lead to subtle problems like consolidation of incoherent views – the type of problems which sometimes pops up in form of the notorious 'generated code – do not modify – modified nevertheless' files. This poses the question, whether conditions for integration of simultaneous development of different (editable) views may be better in the structural approach is worth a second consideration.

Another interesting feature of the structural integration idea is, that it allows for maintenance of an executable interface, e.g. if the model elements implement something similar to the component interface of Microsoft's CTS. This will facilitate providing the domain expert with design time support, something very important in a scenario where the domain expert actually uses model driven development techniques to generate tools needed to build the domain model.

Since greed will become well understood only through doing solution development, it needs to be possible to organize integrated development of solutions and the meta-model of both views and their relations. Also is it more important to support the domain expert in building the domain models themselves than boosting productivity in terms of production of implementation from design – which is a consequence of the somewhat paradoxical situation where domain model development is performed through implementation model development, so that the latter actually becomes a vehicle for the first.

Object oriented vs. grammar oriented integration

The term 'language' is often used in a more general way than that of formal logics and includes graphical languages and general rules for handling and combining symbols [Fo05] [St05]. Nevertheless the question remains, what kind of thing a model element is considered to be with respect to its role in view integration: Is it considered as an instance of a meta-model class? If so, then it can show behavior, manifest itself as an instance of different interfaces or base classes, which exhibit different abstractions, change state, maintain references to other model elements and so on. Or is the model element considered a production of a formal grammar? In this case, it may appear in the input stream of a parser or the output stream of the code generator but cannot show behavior and there will not be a class hierarchy.

Both approaches can coexist and it is possible, to switch between them: a model that exists in form of XML-code may be transformed into a DOM, and a model, that exists as an interwoven lattice of objects may be persisted as a XML file. Yet, even if grammar-oriented and object oriented meta-models coexist, it must be decided which of the two representations will be used for integration of different views. The distinction between object oriented and grammar oriented integration can be summarized in the table below:

Object oriented integration	**Grammar oriented integration**
Model elements are considered instances of meta-model classes for the transformation between views.	Model elements are considered productions of formal grammars for the transformation between views.
The models are general graphs containing of nodes and references. Relations between elements are represented directly by associations or by class-class relations.	Relations between elements are represented indirectly. The algebraic structure of the web of relations extends the set of production rules of the formal grammar.
Shared properties of model elements of both views are represented by object oriented mechanisms, e.g. by deriving classes of both views from a common base class or by realizing views as interfaces of an underlying integrated model.	Shared properties of model elements of both views are represented by relations between the views' languages, e.g. by defining these languages by application of filters to an underlying integrated model language.
Explicit transformations are based on message exchange between model objects.	Explicit transformations are based on parsing, merging and code generation.

Both object oriented and grammar oriented integration techniques are compatible with structural as well as with dynamic integration:

- Object-oriented + structural: Concrete types in the meta-model type system are complete solution types. The domain- or implementation specific types are abstract type or interfaces implemented by the solution type. Shared properties of the domain-

and implementation type are represented as properties of a common base-class of these abstract types.

- Object-oriented + dynamical: In this case the domain- and implementation types are concrete. Domain and implementation specific information is thus represented by different object instances that must have methods for synchronization. Shared properties are either redundant in both views or there are references to specialized shared-property types.
- Grammar oriented + structural: Here the code of the solution is a production of one formal grammar. The languages of the views are defined by homomorphisms from the complete solution language in- or onto the languages of the individual views.
- Grammar oriented + dynamical: The view models are represented in domain- or implementation specific languages, the latter often being a standard language. Coherency of views is assured by parse, code-generation and merge algorithms.

The grammar oriented approaches have a long tradition of mathematical analyses [Ha71] [Ag05]. Formalizations of object oriented approaches have recently been motivated by the demand for a theoretical base for object oriented MDD [OMG01].

Grammar oriented integration yields an explicit representation of relations between views in templates or grammar files. The object oriented alternative represents the relation between views in the class system of the meta-model. This is a trade off between the flexibility and transparency of grammar oriented techniques on the one hand, the power of object oriented concepts and the support for the development of object oriented meta-models – which also are object oriented solutions – by modern IDEs on the other hand.

Discussion

Structural versus dynamic and grammar-oriented versus object oriented MDD-approaches yield four combinations with different advantages and drawbacks.

Object oriented structural integration allows optimal support for the domain expert through interaction with the model elements and helps to integrate synchronous co-evolution of the application and it's meta-model. The most obvious disadvantage is the necessity of strong domain-expert – meta-modeler interaction and continuous participation of the latter in solution development, implied by the definition of domain/implementation-mapping in the Meta model's type system.

Dynamic integration within an object oriented meta-model simplifies the task of addressing different target platforms from the same domain model, compared with structural object oriented integration. The drawback is, that it is more difficult to maintain coherency between domain and implementation meta-model, if neither is stable from the beginning.

Structural integration within a grammar oriented meta-model is theoretically well understood but the concepts of grammar-to-grammar homomorphism and/or tree-rewriting are rather abstract. Tool support for practical work is also less elaborated than modern IDEs for object oriented software development.

Dynamic integration on a grammar based meta-model can increase productivity a lot if a high degree of completeness in code generation can be achieved. The precondition for this is, that the domain meta-model is understood in advance. Also, while it is possible to provide the domain expert with design time support through the development environment, it is

somewhat complicated to integrate functionality provided by the generated implementation model into design time support functionality.

Conclusion

Recalling that the task is to build a development environment for the domain expert within which generated implementations are immediately available to provide them with design time support and that knowledge will be insufficient to define a complete meta model in advance, the preceding consideration suggest a recommendation for object oriented and structural integration. This approach appears to be most suitable for a project like greed, where the domain expert shall be supported with the means to explore their domain with machines that are generated from the ideas that they can formulate in terms of domain specific concepts and where the meta-model continuously has to be adapted to new requirements.

References

Ag05: Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A. *The design of a language for model transformations.* Vanderbilt University, http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2005_The_Design.pdf

Ba91: Bateson, Gregory: *Cybernetic Explanation.* In: Klir, George J. (ed) *Facets of System Science.* ISBN 0-306-43959-X Plenum Press, New York and London 1991

Fo05 Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?* http://martinfowler.com/articles/languageWorkbench.html

Gr04: Greenfield J. Short K., Cook S., Kent S. Crup J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. ISBN: 0471202843 Wiley, 2004

Ha71: Harrison, M. *Introduction to Formal Language Theory*, Addison Wesley, 1978.

Ko05: Kovse, Jernej. *Programmieraufwand mit Generatoren drastisch reduzieren*. Computerwelt, 2005. http://www.computerwelt.at/detailArticle.asp?a=97529&n=2&s=97526

OMG01 Architectur Board ORMSC; Miller, Joaquin & Mukerji, Jishmu (eds): Model Driven Architecture. http://www.omg.org/docs/ormsc/01-07-01.pdf, 2001.

Pa02: Paige, Richard & Ostroff, Jonathan: *The Single Model Principle*. Journal of object technology, 2002. http://www.jot.fm/issues/issue_2002_11/column6

St05: Stahl, Tom & Voelter, *Modellgetriebene Softwareentwicklung*. ISBN 3-89864-310-7 dPunkt, 2005. (English: Model-Driven Software Development, Wiley 2006)

Sw95: Schwager, Jack D. *Schwager on Futures. Technical Analysis.* ISBN: 0471020567 John Wiley & Sons 1995.

To06 Tolvanen, Juha P. Domain-Specific Modeling for Full Code Generation. Software Developer's Journal 2006. http://en.sdjournal.org/products/articleInfo/86