Towards a Generic Architechture for Multi-Level Modeling

T. Aschauer, G. Dauenhauer, W. Pree

Technical Report
August 10, 2009

Software & systems Research Center (SRC)
C. Doppler Laboratory *Embedded Software Systems*
Univ. Salzburg
5020 Salzburg
Austria, Europe

Towards a Generic Architecture for Multi-Level Modeling

Thomas Aschauer, Gerd Dauenhauer, Wolfgang Pree
C. Doppler Laboratory Embedded Software Systems, University of Salzburg
Jakob-Haringer-Straße 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract

This paper presents the architecture of a model-driven engineering framework which relies on the unified notion of classes and objects, as pioneered by SELF *[1]. We implemented this architecture for the domain of testbed automation systems and argue that this architecture can be generalized. We outline why our first prototype implementation following a conventional, UML-like metamodeling approach failed and how the follow-up implementation is aligned with the more appropriate so-called Orthogonal Classification Architecture (OCA). While the OCA has been thoroughly studied theoretically, we applied OCA for a real-world case in the automation system domain. We demonstrate that this modeling approach is feasible and implies a straightforward, clear-cut decomposition of the framework into implementation modules, leading to comprehensible software architectures.*

1. Introduction

For the development of large software intensive systems, model-driven engineering (MDE) is a promising approach for coping with the inherent complexity. The basic idea of MDE is that models are the main artifacts describing a system under study, and that a model at a certain level of abstraction can be transformed into another model at a possibly different level of abstraction. Metamodels play an important role in MDE: they specify the structure of models that are to be processed, i.e. they define modeling languages [2]. For some domains general purpose metamodels are not well suited, so that in practice metamodeling tools are used to define domain specific languages.

Atkinson and Kühne [3] argue that for certain domains the classical approach of metamodeling falls short because it does not allow describing the domain at different levels of abstractions, i.e. at different domain specific meta-levels. Most of the prominent metamodeling languages do not have built-in support for such domain specific meta-levels, and so work-arounds are used in practice [3].

The main contribution of this paper is the presentation of a MDE environment for the specific domain of testbed automation systems, featuring multi-level modeling. We further describe the resulting, significant impacts on the software architecture. By examining an instructive example of this domain, in section 2 we show some problems resulting from employing a conventional metamodeling approach. In section 3 we present the basic idea of the Orthogonal Classification Architecture [4] that clearly distinguishes between two metamodeling dimensions. We further describe how our framework benefits from that approach, and present the corresponding tool in section 4. The impacts on the software architecture are discussed in section 5. A brief presentation of the current implementation and future work is given in section 6. Related work is discussed in section 7, while section 8 concludes the paper.

1.1 Model-driven parameter generation

Testbed automation systems, used for example in R&D of clean technology engines, are inherently complex for various reasons. They are (1) usually built individually of (2) thousands of ready made parts, which are (3) often customized. Testbeds also integrate (4) sophisticated measurement devices that are software intensive systems by themselves. Therefore, the automation system software has to be flexible and highly customizable. Customization is done by providing values for software parameters, such as PID-controller values. In a typical setup they account to tens of thousands of integer, string, and float parameters. Currently these parameters are managed in many different, mostly unstructured, configuration files. Without appropriate software support, testbed customization is an error-prone and time-consuming task.

Our research aims at developing of a framework for model-driven generation of automation system configuration parameters. The idea in essence is to describe all necessary parts of the testbed's structure in a domain specific modeling language. A testbed model comprises all hardware and software parts such that configuration parameters for the automation system can be generated automatically. Figure 1 sketches this concept.

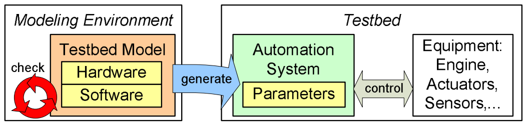


Figure 1. MDE for testbed configuration

Processing of testbed models involves three phases: In the modeling phase a user builds or modifies a testbed model. Once the model is finished, the corresponding automation system parameters can be generated in the generation phase by a separate transformation system. In the execution phase, the automation system is started and a defined sequence of test steps is performed.

In this paper we only consider the modeling aspect and do not deal with other aspects such as the transformation. To illustrate our arguments, we use a typical example from the domain: Modeling an engine and the corresponding sensors.

2. Conventional metamodeling

In state of the art metamodeling tools, the metamodel is explicitly represented and generation or interpretation techniques are used for creating the modeling environment. MetaEdit+ [5] or GME [6], for example, are two widely used tools for creating domain specific modeling environments. Usage of such tools basically involves two steps: a) defining a metamodel that defines the domain specific language, and b) using the metamodel to provide an environment for domain specific modeling. The latter step can be realized either by a generic environment that interprets the metamodel, or by a custom environment generated from the metamodel. For both cases, however, the metamodel appears as "fixed" in the modeling environment, that is, it cannot be modified there, as sketched in figure 2.

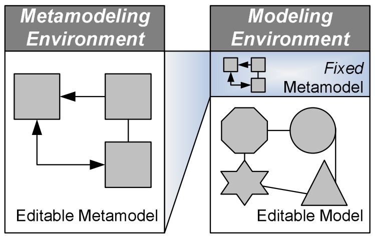


Figure 2. Conventional metamodeling

The metamodeling environment allows for defining the language concepts, such as "Sensor", "Engine", "I/O Device", as well as their properties, such as the sensor's measurement range, or that an engine's speed and inertia. Relations between these concepts such as the fact that engines can contain sensors and sensors are connected to I/O devices are also part of the language and thus are described in the metamodeling environment. For defining additional constraints, these environments usually provide some dedicated language. Besides defining just the metamodel, i.e. the abstract syntax of the domain specific language, these environments typically also allow specifying the visual appearance, such as specific symbols used for representing domain concepts.

Once the metamodel is defined, it can be used for creating the modeling environment. The metamodel controls what domain models can be created and the concrete syntax associated with the metamodel controls the look of the diagrams that can be drawn. In other words, the domain model is an *instance of* the metamodel. The domain model and the metamodel are at different meta-levels, similar to the M1 and M2 levels in UML [7]. To our knowledge, all prominent metamodeling environments in principle follow this approach.

2.1 Hierarchies of model levels

As Atkinson and Kühne [4] show, the limitation to only two modeling levels is not suited well for all domains. As an example, consider the case where the domain model level itself needs to be able to represent both *domain types* such as a manufacturer's family of four cylinder Diesel engines and *domain instances* such as concrete engines being mounted on the testbed. In the context of UML, work-arounds for representing both, types and instances within class diagrams, have been proposed such as the *type object pattern* [8]. Atkinson and Kühne describe in detail why such workarounds lead to accidental complexity [3].

Similarly to the type object pattern, the representation of types and instances within the same model level is also possible in the mentioned metamodeling tools. The rich constraint language they provide can be used to encode the semantics of the artificial instantiation relationship to ensure that only valid models might be built. Analogously to the previous case, this also induces accidental complexity.

Another option is to split the domain model level into a *domain type* model level and a *domain instance* model level. One might use a metamodeling tool to a) create a modeling environment for domain types, such as all the engine families of a manufacturer. The generated metamodeling tool then can be used to b) create the final modeling environment where one creates concrete engine instances. The domain instance model thus conforms to the domain type model, which in turn conforms to the metamodel. So the three models form a linear metamodel hierarchy, as sketched in figure 3.

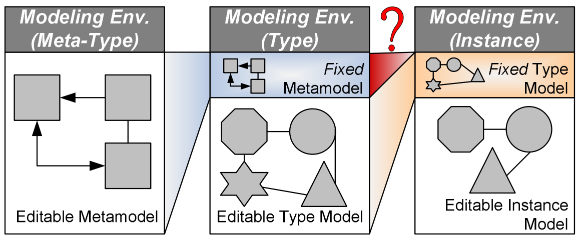


Figure 3. Metamodeling tool chain

Since conventional metamodeling environments do not support multiple model-levels, however, the metamodel can not be explicitly represented in the final modeling environment for domain instances, which is depicted by the asterisk in figure 3. As a consequence, the instance models can not explicitly access the information represented in the leftmost model. So in context of the modeling environment shown at the right hand side, it is for example not possible to determine the circle's meta-type, i.e. the corresponding square. If this information is of importance, e.g. for the transformation system in an MDE-framework, the concepts of the meta-model have to be represented somehow in the type-model, which again is a workaround and leads to the problem of "replication of concepts".

2.2 Our prototypical implementation

Our first implementation of the modeling environment followed the conventional metamodeling principles shown in figure 2. The metamodel was represented in UML and transformed into C#-code using the openArchitectureWare [9] MDE tool. The generated code was then linked with our own implementation of a generic modeling environment. The problem was that our environment had to support both, domain types and domain instances. We sidestepped that issue by simply treating them as *technically the same kind of things*. This solution was inspired by the concept of prototypical programming languages, such as SELF [1].

Figure 4 shows how we translated this idea into our domain. Component is an example of the fixed metamodel elements represented as code in the environment. Such metamodel elements can be instantiated to create initial domain model elements such as Engine. Different kinds of engines may now be either created by also directly instantiating Component, or by *cloning* the initial Engine. Cloning, however, is the preferred way to create new elements, since this way features can be introduced incrementally. In the example, Engine declares two attributes, Inertia and MaxSpeed. Since in the prototypical approach each element is an instance, Engine must provide values for these attributes. Whether these values are reasonable default values or special "null" values does not matter for the formalism, but can cause confusion.

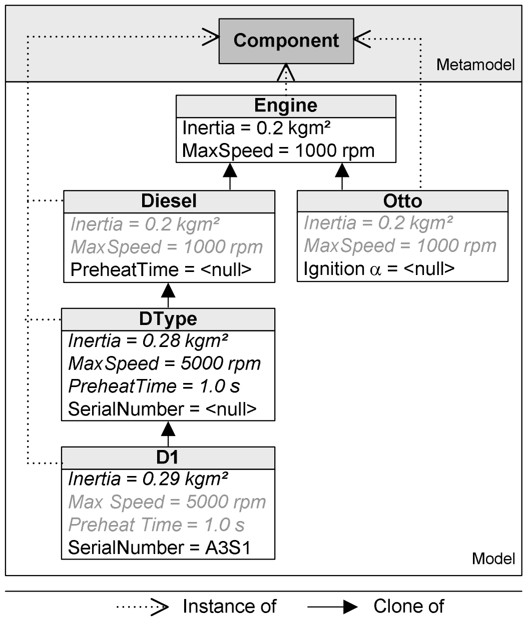


Figure 4. Cloning domain elements

Diesel and Otto represent two kinds of engines; since they are cloned from Engine, they receive copies of the attributes Inertia and MaxSpeed, as well as their values. Italics script is used to mark such copied attributes; grey text is used to express that the attribute values are kept unchanged. DType represents a family of diesel engines. As such, values of attributes copied from other engines can now be made more specific. Inertia for which Engine could only propose a rough guess can now be made more specific; the low value for MaxSpeed proposed by Engine to prevent damage can now be safely raised to the engine's nominal speed. D1 finally is a concrete, physically existing member. Inertia, for which DType proposes a default value, can now be determined.

Necessity for domain types.

In our initial prototypical approach, following the ideas of SELF, cloned elements may be modified without any restrictions. So it is valid to add new attributes, change the values of copied attributes, but also to *remove* them, or change their data type. Our domain, however, often demands for more strictness in the hierarchy of elements, because clients of such model elements may depend on them. For example, the attribute Inertia, which is considered essential for all engines, must not be removed; the value of the MaxSpeed attribute specified for the whole family of DType engines must not be changed for individual DType engines. Figure 5 gives an example of such a client: the code for generating parameters.

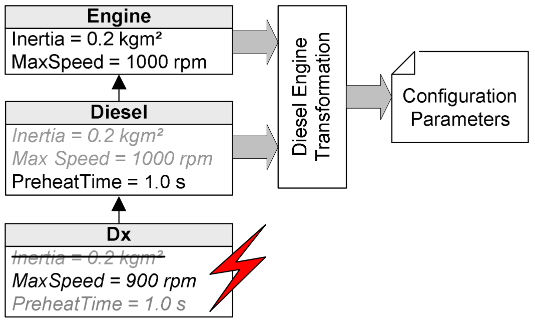


Figure 5. Parameter generation

As described in section 1, the main purpose of our environment is generating an automation system configuration. Thus the transformation code must be able to rely on a correct model in order to generate valid corresponding configuration parameters. Assume that one specific part of the transformation code, responsible for generating diesel engine related parameters, relies on the fact that all diesel engines are engines and as such have an Inertia attribute. If any element claiming to be a diesel engine violates the transformation code's assumptions, no parameters can be generated.

The point is that elements must be able to make statements about other, more specific elements, which actually is similar to a relation between classes and instances. To emphasize this fact, levels should be drawn like in figure 6.

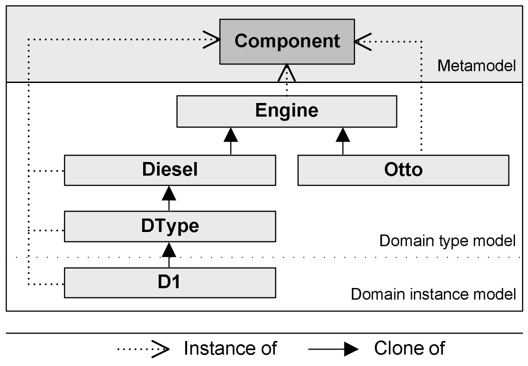


Figure 6. Model levels

A problem is that with the prototype formalism these levels can not be represented explicitly, because the Clone-of relationship could stand for generalization as well as for instantiation. The relation between the elements at the domain type level is that of generalization, as represented by the Clone-of relationship. The relation between elements at the domain type-level and the domain instance level, however, is also modeled by the same relationship. The resulting ambiguity and the resulting problems are analogous as introduced by the type-object pattern [3].

Besides these rather technical problems with a prototypical modeling approach, we realized that we also faced a more severe problem with *perception*. Users of our environment often got confused about what a model element means, i.e. whether it is a domain type, such as a family of engines, or a domain instance, such as a concrete engine, and whether cloning an element means creating a domain type or a domain instance.

3. Multi-Level Modeling

To overcome the limits of conventional metamodeling discussed in section 2.1, multi-level modeling is an alternative approach. The basic idea of multi-level modeling is to explicitly represent the different abstraction levels of model elements in complex domains. Different flavors of multi-level modeling have been proposed. Atkinson and Kühne, for example, propose a uniform notion of classes and objects, also known as a clabject [3], that allows for an arbitrary number of classification levels and whose advantages are well documented in literature [3, 10, 11]. In principle, a clabject is a modeling entity that has a class facet as well as an object facet.

3.1 Clabjects

The example in figure 6 demonstrated that in the testbed domain the representation of elements at different abstraction levels is required. In figure 7 we present how this example can concisely be modeled with clabjects. The notation used here is similar to that of the original clabject concept, that is, a combination of UML notations for classes and objects [3, 12]. Each model element has a compartment for the name, and a combined compartment for the type facet and the instance facet represented by *fields* [3]. The dashed arrows between the levels represent the "instance of" relationship. In this notation inherited or instantiated fields are only repeated when assigned a specific value.

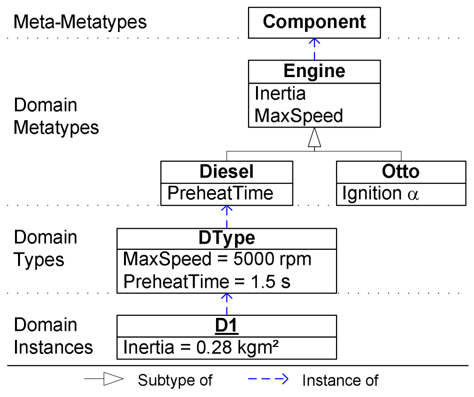


Figure 7. Domain model with clabjects

With a uniform representation of type facets and instance facets, modeling our example is straightforward. At the domain metatype level, clabjects Engine, Diesel and Otto are modeled as a conventional class hierarchy. Their fields, like Inertia and PreheatTime, are part of the corresponding clabject's type facet. Specified at the domain type level, the clabject DType is an instance of Diesel. It provides values for the fields MaxSpeed and PreheatTime, which are part of DType's instance facet. Although not shown in the figure, DType could also introduce a new field, which would then be part of its type facet. The domain instance D1 in turn instantiates DType and provides a value for the Inertia field. Note that D1's type facet is empty. By definition, the clabjects at the top-level only have a type facet, whereas the clabjects at the bottom level only have an instance facet.

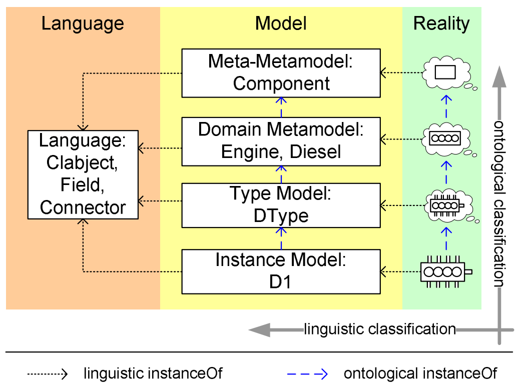
An important property of this model is that we only have one kind of instance-of relationship, rather than the ambiguous relationships as we had in figure 6. Moreover, this relationship is well-defined and has the same semantics at all modeling levels.

3.2 Orthogonal classification

Atkinson and Kühne separate two orthogonal kinds of instantiation, also known as the Orthogonal Classification Architecture [10]: linguistic and ontological instantiation. Linguistic instantiation represents the type-instance relation between a modeling language and model elements. Ontological classification represents the type-instance relation between model elements at different levels, such as the elements of our last example in figure 7.

An illustration of this concept, adapted from [10], is shown in figure 8. Linguistic classification is about the relation between the modeling formalism's implementation, e.g. Clabject on the left hand side, and its instances, e.g. Engine or DType in the middle part. The relation between reality and the model elements is also a linguistic classification. "Reality" here comprises real-world physical elements such as engines, but also conceptual entities as perceived by domain experts, such as the family of diesel engines.

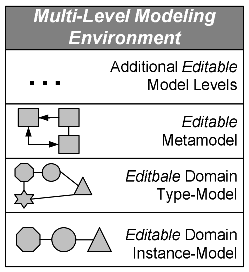
Ontological classification, in contrast, is about the relation between model elements at different levels, e.g. the relation between Diesel and DType, or the relation between DType and D1. It is important to note that our particular example requires only four ontological levels; the number of ontological levels required by another domain might be different.



Language
Model
Reality
Meta-Metamodel: Component
Domain Metamodel: Engine, Diesel
Type Model: DType
Instance Model: D1
Language: Clabject, Field, Connector
ontological classification
linguistic classification
linguistic instanceOf
ontological instanceOf

Figure 8. Classification dimensions

Tool support.
Separating the linguistic and the ontological classification dimensions has immediate effect on the implementation of a modeling environment. The environment provides the implementation of the modeling language, i.e. the linguistic classification of modeling elements. Because the ontological dimension is orthogonal, the modeling environment can immediately handle multiple ontological classification levels, as shown in figure 9.



Multi-Level Modeling Environment
...
Additional *Editable* Model Levels
Editable Metamodel
Editable Domain Type-Model
Editable Domain Instance-Model

Figure 9. Multi-level tool

The figure illustrates that due to the explicit definition of the ontological instance-of relation, an arbitrary number of ontological modeling levels can be supported by a single tool. The relationships between these layers are concisely captured. The linguistic metamodel, which is not shown in the figure, is orthogonal to the ontological model levels. Thus, adding additional ontological meta-levels is straightforward, which enables tools to support a wide range of application domains. Although the different meta-levels have to be editable in principle, they are not necessarily editable at all times. Nevertheless, all metalevels are represented and accessible at all times, which for example is beneficial for building model transformations.

4. Multi-level MDE tool

In a second version of our model-driven parameter generation framework, we employed the multi-level modeling approach as presented in the previous section. Here we present the system's architecture from the developer's perspective.

The basis for multi-level modeling is the separation of the two classification dimensions. In our domain, the model layers are as shown in figure 8. For the architectural decisions on how to decompose the system into implementation modules, the multi-level hierarchy also plays a central role: The two classifications dimensions not only define how the model is organized, but also define the dependencies of implementation modules. Figure 10 shows an example.

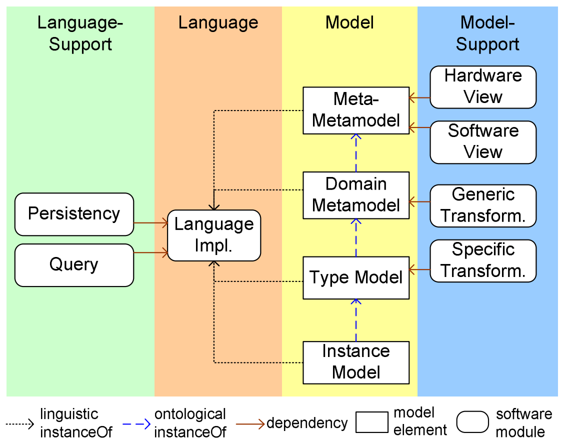


Figure 10. Module alignment

The implementation modules comprising the MDE-tool are arranged according to the two classification dimension. The *Language Implementation* module contains the C# implementation of the core language constructs such as Clabject, Field, or Connector. The modules on the left hand side depend on these language concepts only. The modules on the right hand side, in contrast, depend on the language implementation and also on concepts defined at different metamodeling levels. The software modules implementing the rendering of hardware and software aspects, for example, depend on the Meta-Metamodel since there the corresponding model elements, such as Sensor or Dataflow Signal, are defined. These generic visualization modules, however, are independent of more domain specific concepts such as Engine or Diesel. But we also can identify implementation modules that rely on exactly these concepts, e.g. the transformation module for generating automation system parameters. In fact, as motivated in figure 5, there are two such transformation modules: one that implements transformation at the level of the domain metatypes, and a second one for specific transformations of certain domain types, such as DType.

4.1 Language module

The linguistic level of a multi level modeling environment must be capable of representing arbitrary ontological models at runtime. Thus an appropriate linguistic metamodel has to support the ontological model entities, their properties, and relationships between them. Furthermore, multiple ontological levels have to be supported, so the abstractions in the linguistic level must provide a means of expressing instantiation and generalization relations. The entities of our language module basically are Clabject, Connector, Field, and Data Type, as shown in figure 11. Note that we omit certain implementation details.

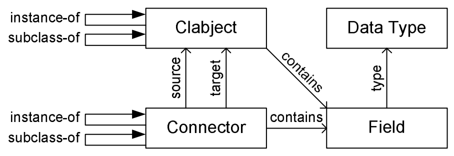


Figure 11. Language module entities

Clabjects represent model elements which may either be types or instances. As described in section 3.1, a clabject can be an instance of another clabject or it can be a specialization of another clabject. Clabjects are further described by their contained fields, which can represent either type or instance specific information. Each field has a data type, which may either be a predefined primitive data type such as integer or text, or a user defined type such as a list or record. As with clabjects, connectors also can represent types or instances, and connectors can be specializations of other connectors. Properties of connectors, such as multiplicities, are further described by their contained fields.

The linguistic level manages arbitrary hierarchies of ontological models in memory, represented by these basic abstractions. It does not interpret the models beyond enforcing constraints, such as multiplicities of connectors. In other words, the linguistic level is *neutral* to the models it holds, or domain independent.

4.2 Language support modules

Besides providing modeling primitives, the linguistic level is also concerned with managing models. A typical function implemented in a *language support module* is, for example, persistency of models. In our case this includes functions for loading and storing model elements and a library mechanism that allows modularizing the model space and enables reuse of compound model elements. Another typical function is a query mechanism for searching the model space.

4.3 Model support modules

Ontological model elements represent the domain concepts and entities. In realm of an MDE approach, the implementation of certain functionality depends on these domain abstractions, such as the transformation code for automation system parameters. The implementation of a *model support module* relies on the structure of the corresponding ontological model level.

An interesting property of multi-level modeling is that every ontological model layer in fact defines a set of domain abstractions which form domain specific language. A generic framework thus could render the domain entities with different, custom defined symbols. The infrastructure proposed by Atkinson et al. [13] uses this approach. In our tool we take this idea one step further and allow model support modules to provide a completely custom visualization of domain entities. The Hardware-View and the Software-View modules as shown in figure 10, for example, define very specific visualizations for subsets of the model entities. Support modules for lower ontological levels may provide a further customized visualization, such as a specific symbol for a diesel engine.

4.4 Separation of concern dimensions

The presented alignment of implementation modules along the two orthogonal classification dimensions provides a straightforward separation of modules with different concerns. Modules implementing the language and supporting functions are at the linguistic dimension, while modules implementing model-dependent features are at the ontological dimension. So we get a clear separation between models that are domain independent, which can be reused among domains, and modules that are domain specific, which can be reused within a domain, e.g. for software product lines [14].

This particular alignment also defines a certain pattern of module dependencies. The linguistic implementation modules are independent of the ontological implementation modules. The converse, however, does not hold since the ontological implementation modules might need to access the functions of the linguistic implementation modules. The software view module, for example, uses the query module to traverse model elements. As a consequence, an ontological implementation module can make use of on any linguistic implementation module and also any ontological implementation module at a higher level. A linguistic implementation module in contrast can make use of other linguistic implementation modules only.

This separation of concerns, which now clearly separates domain specific concerns from general concerns, can also be used as a measure of where to implement new functions. Assume, for example, that we want to extend the multi-level MDE tool with undo/redo functionality. Since undo/redo clearly is independent of our target domain, it has to be implemented along the linguistic classification domain, i.e. in figure 10 it has to be placed next to the Persistency and Query modules.

5. Architectural impacts

A multilevel modeling architecture has both technical as well as organizational impacts. Each ontological model level represents a different level of abstraction. These model levels define a clear-cut partitioning of the implementation modules. However, they also serve as a means for communication among the developers and users responsible for different levels, so each user group can talk in terms of their own abstraction.

5.1 Immediate architectural benefits

After the implementation of the multi-modeling kernel, we ported certain business functionality from the first version. By examining the module alignment we could verify whether the functions were implemented at the correct abstraction level. The old undo/redo mechanism, for example, was spread over different ontological model levels. Thus in the new version we factored out the mechanism and implemented it as a language support module. In a later architectural review this turned out to have positive effects due to enhanced encapsulation.

To utilize the power of the compiler's static type checking facility, we provide a way of creating type information from ontological model elements in order to allow checking the corresponding modules. After aligning the logical software architecture along the classification dimensions, we discovered that this mechanism was implemented specifically for the higher model levels, but that it actually is independent of the ontological classification. Implementing the generator for type-information as a language-support module, which is planned in near future, will allow all model support modules to benefit from static compiler checks.

Checking the implementation against the dependency rules described in section 4.4 also revealed a subtle flaw in the module dependency graph: one of the language-support modules depended on the top ontological model level. This demonstrated that the explicit definition of a dependency pattern is indeed a beneficial tool to ensure that the implementation does not violate the defined architecture.

5.2 Splitting model layers

The ontological metamodeling approach allowed us factoring out common behavior that follows a certain pattern. Figure 12 shows clabject diagrams for two sample cases: a) expresses that a Component can contain other Components as well as Sensors; b) expresses that a Component can have a Hardware Drawing, on which Widgets are rendered, which in turn represent components.[1]

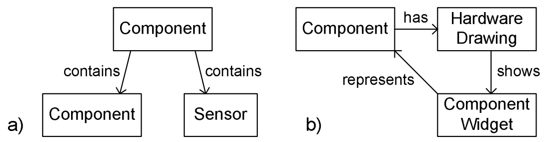


Figure 12. Meta-metamodel fragments.

The examples stand for two general patterns shown in figure 13: the fact a) that a model element can contain other model elements is rather trivial. The visualization pattern b) is motivated by the fact that hardware as well as software aspects are eventually represented as boxes with ports, connected by lines on multiple drawings.

Component and Sensor are instances of the generic Element, and the "contains" connector between Component and Sensor is an instance of the generic "contains" connector defined for Element. Analogously Hardware Drawing, Component Widget and the "shows" connector between them are instances of the generic visualization concepts.

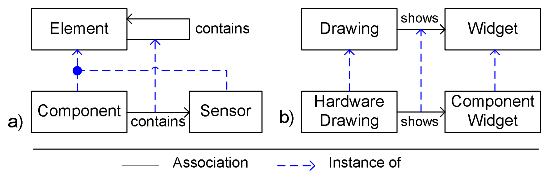
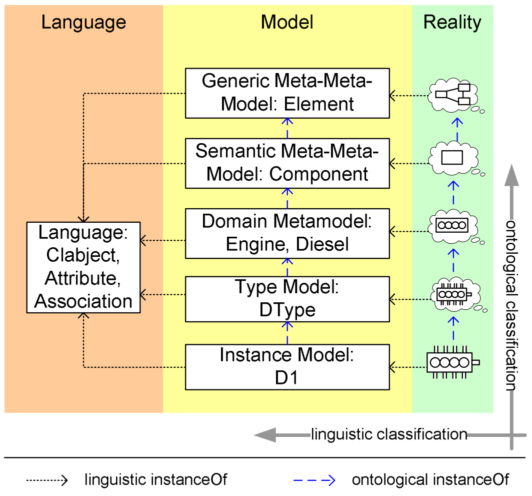


Figure 13. Meta-metamodel patterns.

Introducing generic components for expressing such patterns is not an end in itself; instead it allows us to implement certain aspects of the meta-metamodel support module in a more generic way. Pattern a) is used to visualize the containment hierarchy of all possible models in a tree view. Pattern b) is used for example for managing multiple drawings associated with a component. These functionalities can be implemented without knowledge about Component, Sensor, Hardware Drawing or Component Widget. It is thus an additional ontological level, as shown in figure 14.


Language
Model
Reality
Generic Meta-Meta-Model: Element
Semantic Meta-Meta-Model: Component
Domain Metamodel: Engine, Diesel
Type Model: DType
Instance Model: D1
Language: Clabject, Attribute, Association
ontological classification
linguistic classification
linguistic instanceOf
ontological instanceOf
Figure 14. Split meta-metamodel.

Analogously to splitting the meta-metamodel level, other model levels could be split. Ontological levels above and below, their support modules, and the language and its support modules need not be touched.

5.3 Product families

Ontological levels help creating families of similar products from a shared model and implementation code base. Suppose we have to create multiple modeling environments to support different testbed automation systems, provided by different vendors. Chances are

[1] We use clabjects not only for the model elements, but also for describing the visualization of models. The real visualization pattern however is more elaborate.

that these environments share not only the meta-metamodel, but also the domain metamodel for engine testbeds. The different environments however are likely to require different type models, containing for example different I/O devices.

The same principle holds if we need to create yet another modeling environment targeted, for example, at factory automation systems. Chances are that both testbed automation systems and factory automation systems have enough in common and thus can share the same meta-metamodel and its associated implementation code. They will however not share the same domain metamodel. Figure 15 shows an example.

The example highlights the product family member "Testbed Automation System 1", which uses a set of common elements, shared among different testbed automation systems. These similarities among modeling environments thus define *product families* or *product lines*. In other words: a range of different modeling environments can be produced simply by combining models and their implementation modules.

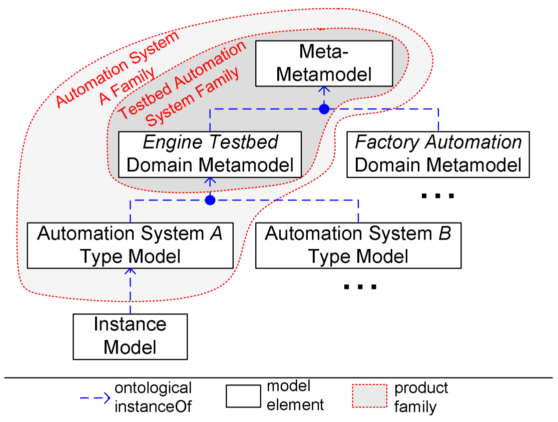

Automation System A Family
Testbed Automation System Family
Meta-Metamodel
Engine Testbed Domain Metamodel
Factory Automation Domain Metamodel
...
Automation System A Type Model
Automation System B Type Model
...
Instance Model
ontological instanceOf
model element
product family

Figure 15. Product family.

6. Status of implementation

Our MDE tool currently exists in its second version, which in contrast to the first version features multi-level modeling. Among the features as presented in section 4, the following are implemented: the modeling language kernel, supplemental modules such as persistency, the ontological layering and the top domain model levels, and the model support modules for visualization of the domain specific language. A generic visualization based on the linguistic level, i.e. a generic clabject view, is not yet implemented.

We are in the fortunate position that an industry partner provides resources for an empirical study. Domain experts currently use the system to model different kinds of real testbeds. These experiments provide strong evidence that modeling testbeds in the way we presented in section 3 allows for a concise description. Another separate group is in parallel working on the corresponding transformation engine.

7. Related work

The idea of unifying classes and objects has a long tradition in the context of object-oriented programming languages, namely in prototype-based languages such as SELF [1]. A SELF-object consists of named slots that can carry values, which in turn are references to other objects. SELF uses only one relationship, the "inherits from"-relationship [1, table 1] that unifies instantiation and specialization. Our first implementation was inspired by that concepts, but the lack of a type hierarchy turned out inadequate for our domain, and this led to the application of multi-level modeling.

Various metamodeling environments are available today, such as the Generic Modeling Environment [6], the Eclipse Modeling Framework [15], or the commercial solution MetaEdit+ [5]. Although built for different purposes, their meta-metamodels are comparable. Nevertheless, none of the three mentioned environments directly supports multi-level modeling. Gutheil et al. point out that these tools can operate within an implicit multi-level framework by building a tool chain [11]. As already outlined in section 2, this does not allow for accessing meta-information except for the immediate metamodel, and thus does not support a uniform treatment of the ontological model levels.

Atkinson et al. [13] propose a prototypical framework for language engineering also based on the Ontological Classification Architecture [10]. The authors present the basic difficulties of implementing a multi-level modeling environment and describe the benefits in terms of two small case studies. They also describe a generic visualization algorithm that is capable of representing domain entities in the form of custom symbols. Furthermore, they argue for the need of a constraint language that is aware of the different ontological levels. Overall, their work builds a foundation for experimenting with the various aspects of building a multi-level modeling tool.

Our work, in contrast, aims to bridge the gap between theoretical work and industrial applications. As such, it on one had is tied more closely to the application of MDE for automation systems. On the other hand, however, this demonstrates that the rather theoretical discussion of the benefits of multi-level modeling can have real impacts in large applications. Moreover, since we not only studied the modeling environment, but the whole MDE architecture, we were able to identify the architectural impacts of the system as a whole and demonstrate the resulting benefits.

Analyzing dependencies in large software system as a measure to manage the architecture is often used. Murphy et al. [16], for example, propose a method to reverse-engineer the so-called "reflexion model", which is then compared to a so-called "high-level model". Our proposed MDE architecture includes a coarse-grained pattern for the dependencies of modules, i.e. it includes such a high-level model. Dependency extraction and management approaches can be used to check the actual implementation's conformance to the dependency pattern, and further to manage the fine-grained dependencies not covered by the pattern.

The MDE-system as presented here is of course only one view on the architecture. In terms of Kruchten's "4+1 View Model" [17], this roughly corresponds to the development view.

8. Conclusion

Multi-level modeling is a powerful approach for concisely describing complex domains where conventional metamodeling is inadequate. Although the approach's advantages are well known, it is not yet widely used in industry. In this paper we presented the application of multi-level modeling in the domain of testbed automation systems. We showed why conventional modeling is insufficient for our MDE requirements and how multi-level modeling can solve the fundamental issues. The lessons learned led to interesting conclusions regarding the overall system architecture. We showed how the principal idea of separating linguistic and ontological classification not only builds the foundation of multi-level modeling, but furthermore provides a reasonable framework for aligning the logical architecture of the whole MDE-system.

Our particular implementation undoubtedly is highly influenced by the peculiarities of the target domain. The conclusions drawn, however, can be generalized since they are only based on the assumptions of the multi-level modeling principles, e.g. the separation of the two classification dimensions. Thus we are confident that our work clearly demonstrates that applying these principles, which have been around for quite some time, in industrial practice is appropriate for representing the domain and also beneficial for the overall system architecture. As such this work can pave the way for applying multi-level modeling to a broader range of applications.

9. References

[1] D. Ungar and R. B. Smith, "SELF: The power of simplicity", *Proceedings of OOPSLA '87*, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 227–242.

[2] D. Gasevic, N. Kaviani and M. Hatala, "On Metamodeling in Megamodels", *Proceedings of MoDELS'07*, LNCS vol. 4735, 2007, pp. 91–105.

[3] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models", *Software and Systems Modeling*, vol. 7, no. 3, 2007, pp. 345–359.

[4] C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling", *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds. LNCS vol. 2185, 2001, pp. 19–33.

[5] MetaCase, MetaEdit+, http://www.metacase.com/mwb/

[6] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The Generic Modeling Environment", *Proceedings of IEEE Workshop on Intelligent Signal Processing*, Budapest, 2001.

[7] Object Management Group, Unified Modeling Language Infrastructure, version 2.1.2, OMG document formal/07-11-04, 2007.

[8] R. Johnson and B. Woolf, "Type object", in: R. Martin, D. Riehle, F. Buschmann (Eds.), *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

[9] openArchitectureWare Website, retrieved April 2009, http://www.openarchitectureware.org.

[10] C. Atkinson and T. Kühne, "Model-Driven Development: A Metamodeling Foundation". *IEEE Software*, Vol. 20, No. 5, pp. 36-41, IEEE Computer Society, 2003

[11] M. Gutheil, B. Kennel, and C. Atkinson, "A Systematic Approach to Connectors in a Multi-level Modeling Environment", Proceedings of the 11th international *Conference on Model Driven Engineering Languages and Systems*, LNCS vol. 5301, Springer-Verlag, 2008, pp. 843–857

[12] Object Management Group, Unified Modeling Language Superstructure, version 2.1.2, OMG document formal/07-11-02, 2007.

[13] C. Atkinson, M. Gutheil, B. Kennel, "A Flexible Infrastructure for Multi-Level Language Engineering", *To appear*, 2009

[14] P. Clements, L. Northrop, and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[15] Eclipse Foundation, Eclipse Modeling Framework Project, http://www.eclipse.org/modeling/emf/

[16] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models." *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM, New York, NY, 1995, pp. 18-28.

[17] P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, vol. 12, no. 6, 1995, pp. 42-50.