Multi-Level Modeling for Industrial Automation Systems

T. Aschauer, G. Dauenhauer, W. Pree

Technical Report
July 24, 2009

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

Multi-Level Modeling for Industrial Automation Systems

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

***Abstract*—Model-driven engineering of software intensive systems requires adequate means for describing their essential properties. For the domain of testbed automation systems, conventional modeling formalisms fall short due to the inadequacy of a fixed meta-level hierarchy. In this paper we identify the core problems by examining real-world examples. As a solution, we propose using a unification of classes and objects, known as clabjects. We propose extensions to the basic clabject notion for handling connector inheritance and instantiation, which are essential for bridging the gap between theoretical foundations and industrial applications.**

Keywords – multi-level modeling; clabjects

I. INTRODUCTION

Testbed automation systems, so-called *testbeds*, are specific automation systems for operating combustion engines under controlled conditions, e.g. for research and development of clean technology engines. The goal of our research group is to contribute to this domain by providing a framework that employs a model-driven approach to the generation of automation system configuration parameters. Section I.A briefly introduces the domain and describes how our modeling framework fits in.

In section II we investigate challenges of the technical representation of domain entities in languages featuring only a fixed number of meta-levels, as e.g. UML [1] or SysML [2] do. We describe a uniform notion of classes and objects, also known as *clabjects* [3], which allows for an arbitrary number of classification levels. Although its advantages are well documented [3, 4, 5], multi-level modeling with clabjects has been barely applied to real-world applications. By describing how our subject domain can benefit from clabjects, we give a concrete industrial use case for this notion.

To apply the approach in an industrial setting, we had to extend the clabject-notion with aspects such as connector inheritance and gradual instantiation. These concepts, described in section III, represent the main contribution of this paper. Since the testbed domain exemplifies a whole class of industrial applications, we argue that a formalism leading to concise models contributes to model-driven engineering in domains with similar requirements. In section IV we discuss related work, while in section V we describe the status of our implementation that is targeted at bridging the gap between the current rather theoretical discussions of multi-level modeling and industrial applications, and we conclude the paper.

A. Model-driven parameter generation

Testbed automation systems are inherently complex for several 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 is highly customizable by software parameters. In a typical setup they account to tens of thousands of integer, string, and float parameters, currently managed in several rather unstructured configuration files.

Without appropriate software support, customization is error-prone and time-consuming. Our research aims at developing a framework for model-driven generation of automation system configuration parameters. This requires a testbed model comprising all essential hardware and software parts, as sketched in figure 1.

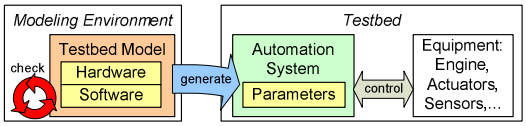

Figure 1. Model processing

Processing of testbed models involves three distinct phases. (1) In the modeling phase a user builds a testbed model. (2) Once a model is finished, in the generation phase the automation system parameters are generated by a separate transformation system. (3) In the execution phase the automation system performs sequences of test steps. The first two phases happen during modeling time, while the last phase happens during execution time. Checks in the modeling phase ensure that only valid models are used for parameter generation.

II. TESTBED MODELING CHALLENGES

For the goal of concisely modeling testbeds, we first need to find appropriate language mechanisms. In the following we derive these mechanisms by studying a typical example: modeling an engine and the necessary sensors to measure certain aspects such as temperature and pressure.

A. Extensibility of types

The first requirement captures the need for the creation of new types at any time during modeling, which includes both, during initial development of models, but also after the testbed automation system has been delivered to a customer. Consider for example the UML class diagram shown in figure 2 that depicts a possible model for engines.

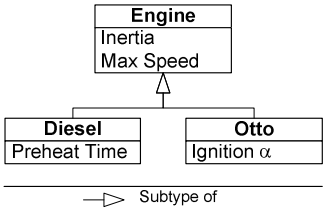

Figure 2. Basic engine class hierarchy

The transformation rules for generating the configuration parameters for the automation system are defined according to the class hierarchy, but it is also necessary to model all instances explicitly since they represent the corresponding real-world entities. Consider for example a rule that traverses the model to find all Engine instances in order to generate the configuration parameters corresponding to Inertia and Max Speed. A second rule might afterwards traverse the model to find all Diesel instances and update the generated configuration data to reflect the effect of the Preheat Time parameter.

The UML object diagram in figure 3 shows the corresponding objects for our example, called InstanceSpecifications in UML 2 terminology [1]. Given this model, the transformation rules described above can generate the corresponding configuration data.

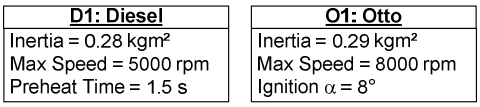

Figure 3. Basic engine objects

Next we consider the case when the testbed and its model are delivered to a customer. Further assume that the customer develops a new type of engine, which is neither a Diesel nor an Otto engine, but e.g. a Wankel engine or an electric engine. Because the class hierarchy above does not support specific attributes of such engines, the customer has to extend the class diagram, as for example shown in figure 4.

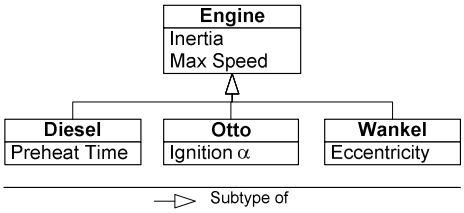

Figure 4. Extended engine class hierarchy

Now it is possible to generate the automation system parameters for a test run of a Wankel engine, given that the transformation systems is extensible in some way such that the new attribute can be processed correctly.

From the example we can see that a fixed class hierarchy of the domain, however thorough engineered, has limited value: Testbeds typically are subject to continuous modification and extension, so we have to support the definition of new engine types, or any other types in general, to support extensibility even after the system has been delivered. This is different to conventional model-driven engineering, where the class hierarchy is designed by programmers and users can only create objects.

To overcome this limitation, certain workarounds and patterns exists, such as the type object pattern [6]. In principle, these solutions emulate classes at the object level as "type objects", thus enabling users to create type-objects, objects, and isTypeOf-relationships dynamically. Although a practical solution, such ad-hoc approaches introduce unnecessary complexity [3].

As we will see in detail later, to prevent the accidental complexity induced by such workarounds we did not stick to UML, but instead introduced our own language that supports creation of new types at runtime.

B. Multiple modeling levels

When analyzing the domain further by looking at our example, we find that for an appropriate description more than two modeling levels are needed. Assume the class hierarchy introduced in figure 2. In a typical case, a customer manages different engine type series with common properties. For each engine of a certain series, some properties are fixed, e.g. the engine kind defining the fuel type as either Diesel or Otto, and the engine's maximum speed.

Assume that a customer defines such a specific Diesel engine type series, called DType. The corresponding model element can be considered a *domain type*, since concrete engines of that type series are instances of DType. DType, which e.g. specifies a certain value for Max Speed, in turn is an instance of Diesel. So the elements Engine and Diesel are *domain metatypes*. Atkinson and Kühne [4] separate two orthogonal kinds of instantiation: linguistic and ontological instantiation. The former represents the relation between a modeling language and model elements, which in our case corresponds to the relation between the language implementation in C#-classes and the C#-objects representing model elements. The latter represents the relation between model elements at different levels. The identified types and instances in our example are an example of ontological modeling, and modeling-levels are the domain metatype level, the domain type level, and the domain instance level.

1) Static attributes.

Modeling our example in UML is not straight-forward because this language does not support domain meta-levels. Nevertheless we can use static attributes as a workaround, so that they represent attributes common to all instances of a class. The corresponding class diagram defining the engine series DType is shown in figure 5. To emphasize the different meta-levels, we highlight the level boundaries by dotted lines.

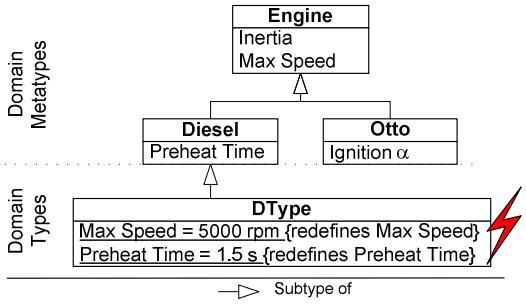

Figure 5. Engine with static attributes

As the Max Speed and Preheat Time attributes are defined in superclasses of DType, we need to specify the corresponding attribute values. UML allows redefinition of elements [1, p. 77]. The redefined element and the redefining element have to be compatible in some way not further specified. In UML terminology this is called a "semantic variation point".

The class diagram in figure 5, however, is problematic since class Engine specifies Max Speed as a dynamic attribute, i.e. individual to each instance, while DType redefines this attribute as a static one, i.e. common to all instances, also known as class attribute. If this attribute is always type-specific, this could be regarded as a flaw in the original class design of Engine and Diesel. Earlier in our example, however, we had instances that specified this attribute individually, e.g. in figure 3, so we have to be able to support both cases.

2) Uniform representation of classes and objects.

In the context of domain models, Atkinson and Kühne define accidental complexity as introduced due to mismatches between the problem and the technology to represent the problem [3]. They argue that modeling languages that allow using only two levels, such as UML with the class and object level, induce accidental complexity when modeling domains that inherently require more modeling levels. Atkinson and Kühne call this the "level mismatch problem". The root of the problem is that at least two domain levels are mapped onto the same model level. The authors also offer a solution: the concept of a *clabject*, a modeling entity that has a type facet as well as an instance facet.

As we can see from our example, the difficulty of representing the domain appropriately with UML mechanisms results from the fact that the domain requirements cannot be directly modeled in this language. We can further discover that the domain features at least three ontological levels. As a consequence, the level mismatch problem arises when trying to use UML to model the domain. To prevent the resulting accidental complexity, we use our own modeling language based on clabjects. Note that since appropriate support for users is essential, we provide a modeling environment that supports the creation of types and instances as well. Figure 6 shows how our example can be modeled with clabjects.

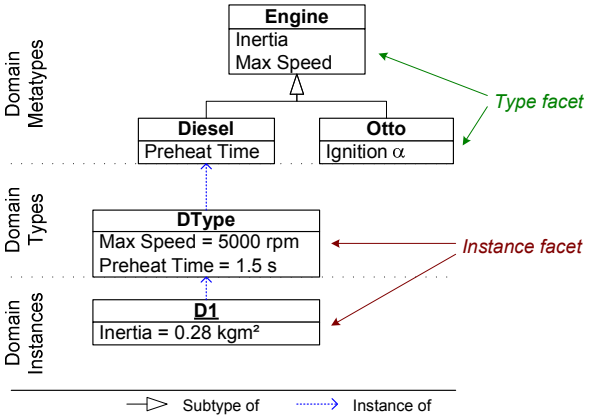

Figure 6. Engine hierarchy with clabjects

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, 7]. Each model element has a compartment for the name, and a combined compartment for the type facet and the instance facet. The unification of attributes and slots is called *field*. The dashed arrows between the levels represent the "instance of" relationship. With this uniform representation of type facets and instance facets, our example can be modeled concisely. 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.

III. Extended Clabjects for Testbed Models

In the previous section we have seen that clabjects are suitable for concisely modeling the engine example. As we will see in the sequel, however, this is not sufficient for describing a complete testbed, so we have to extend this notation to get concise testbed models.

A. Strictness

One requirement of the example that has not been addressed so far is that we still have to be able to create direct instances of Otto and Diesel engines, such as O1 and D1 of figure 3. This raises the question of how strict we separate our modeling-levels. The strict metamodeling doctrine [4] allows only instance-of relations to cross level-boundaries; each element at a certain level must be instance of exactly one element at the level above.

According to the strict metamodeling doctrine, we have to introduce some artificial domain type, say OT1, which is only instantiated once by O1. To avoid such replication of concepts, Gitzel et al. propose a relaxed definition of strictness that allows instance-of relations to cross more than one level boundary [9]. Employing this definition, we can model the example as shown in figure 7.

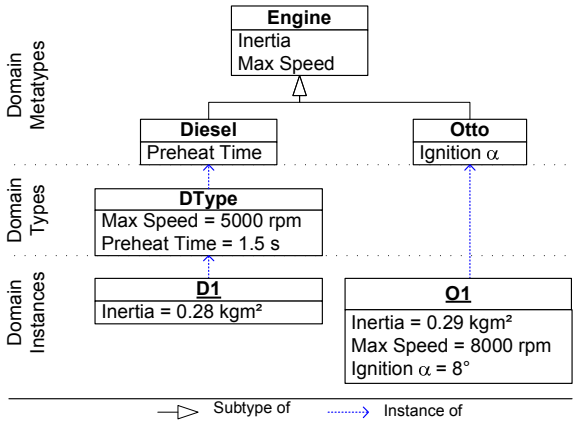

Figure 7. Relaxed strictness

With the relaxation of strictness, however, we loose the unique identification of a clabject's meta-level. To resolve that issue, our clabjects are explicitly assigned to the non-strict modeling-levels. These levels are a pragmatic compromise to the meta-levels as implied by the strict metamodeling doctrine, which implicitly still can be observed in our approach.

B. *Field redefinition*

Some modeling languages such as UML [1] feature the redefinition of attributes. Assume for instance that the field Inertia of Engine in figure 6 has a domain of (0.0[kgm²]…1.0[kgm²]). Further assume that for each engine of the type series DType the inertia is within stricter bounds. When the modeling language permits, we can redefine the field with that restriction. For type-safety reasons, however, in our language we require invariant field types [8].

Invariance of fields is necessary since at runtime a subsystem might refer to the engine D1, which is of dynamic type DType, as an engine of static type Engine. The static type Engine would allow specifying an Inertia value that is not within the stricter bounds of DType, thus possibly leading to runtime errors. Because testbeds are safety critical systems, that is, a failure might cause severe damage of the system or injuries of personnel, we have to prevent such situation by design.

C. *Connectors*

Analogously to clabjects, relationships between elements, called *Connectors* [5], also have a type and an instance aspect. Figure 8 shows the domain meta-type level with engines types and sensor types. The connectors are introduced at an additional meta-meta-type level, which also captures domain concepts.

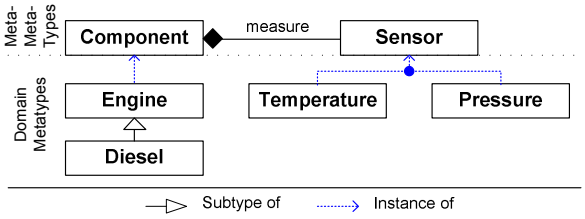

Figure 8. Connectors between elements

The top level elements Component and Sensor as well as the connector “measure” represent generic concepts in the automation system domain. This level expresses that components such as engines may be equipped with sensors used to measure physical phenomena. The top level is instantiated in the domain meta-type level to introduce the element *types* commonly used in the domain of testbeds such as common engine types as well as typical sensor types. Connectors defined between clabjects are also valid connectors for their instances: since on one hand Diesel is a subtype of Engine, and Engine is an instance of Component, and on the other hand Temperature is an instance of Sensor, Diesel can contain any instances of Sensor, including Temperature sensors.

Connector properties such as multiplicities can also be instantiated or specialized, as shown in figure 9. DType is a specific engine type whose instances must always be equipped with at least two TType temperature sensors. This can be expressed by instantiating the connector “measure” and by providing specific values for the multiplicity. Note that instantiation of the connector “temp” does not affect other elements: for example, DType engines may still contain PType pressure sensors, since the generic connector “measure” is still in effect. The semantics of instantiation is that the connector type “measure” can still be used for DType, but it is constrained by the connector instance “temp”, i.e. it has to connect at least two TType instances.

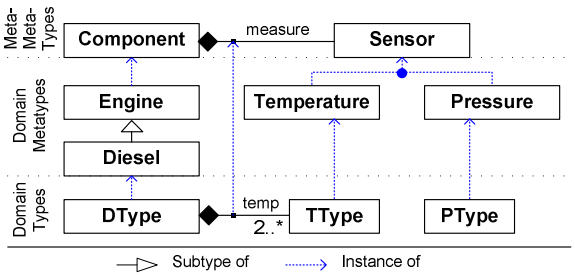

Figure 9. Connector instantiation

Connectors must be instantiated and specialized in a compatible way: bounds for multiplicities must not be widened and the corresponding clabject types must be compatible. In our example: “measure” has unbounded multiplicities on both ends, the instantiated connector “temp” has equal bounds on the DType side, but narrower bounds on the TType side. DType is an instance of Diesel, which in turn is an instance of Component, and similarly, TType is an instance of Temperature, which is an instance of Sensor. As motivated above, we assume the structure of the model to be fixed at execution time, i.e. although field values can be modified at any time, connectors and connector links can be altered only at modeling time. This distinction is essential since it ensures that connector instantiation and specialization can safely be *covariant*: clients using the “temp” connector of a DType instance can rely on the fact that all connected elements are TType instances. At execution time, no client can use the “measure” connector to add arbitrary Sensor instances to the “temp” connector, which would necessarily raise a runtime error. During modeling time, the environment can detect the dynamic types of elements to prevent modeling errors. As already described in section III.A, we do not enforce strict metamodeling for clabjects, so the instantiation of “measure” as “temp” is also valid in this respect.

1) *Multiple specialized connectors.*

Connectors can be instantiated or specialized several times to yield multiple distinct connectors even between the same types, as shown in figure 10.

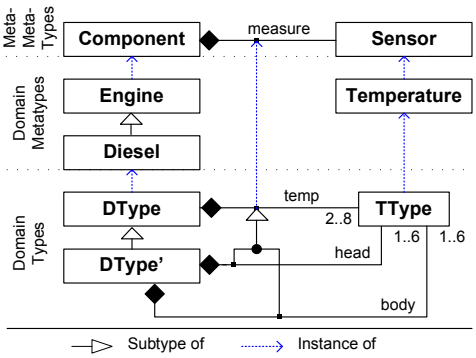

Figure 10. Multiple connector specialization

For example, the connector “temp” can be specialized into two connectors “head” and “body” for partitioning the

set of sensors at the engine's cylinder head and body. Here TType sensor instances may be assigned to the connector "head" or "body". Since "head" and "body" are compositions – indicated by the diamond – a sensor may be contained in only one of them. Composition connectors implicitly define an exclusive-containment constraint, similar to UML compositions [1, p. 113]. Clabjects nevertheless can be assigned to multiple non-composition connectors.

Checking multiple instantiated or specialized connectors for correct multiplicities involves a) checking the individual multiplicities of the connector instances or specializations and b) checking the multiplicity of the connector type or generalization. In the example, both connectors "head" and "body" may contain up to six sensors, although the total number of sensors must not exceed eight, as defined by the connector "temp".

2) Instantiation of connectors.

Figure 6 shows how types and instances of clabjects are described in a single model. Similarly, figure 11 shows how this concept is extended to describe connectors and connector links. Note that here we emphasize the instance facet by drawing connector links.

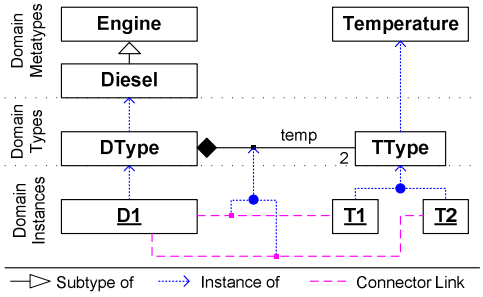

Figure 11. Instantiation of connector

Consider a DType instance D1 whose connector "temp" demands for two TType instances. Analogously to UML, the connector links to sensor instances are instances of the connector "temp", which in our example link T1 and T2 to D1.

3) Pre-allocating connector links.

We already identified the need for non-strict metamodeling. Now we present another requirement that would be impossible with strict metamodeling. Figure 12 shows an example where types and instances of them co-exist *at the same modeling level*.

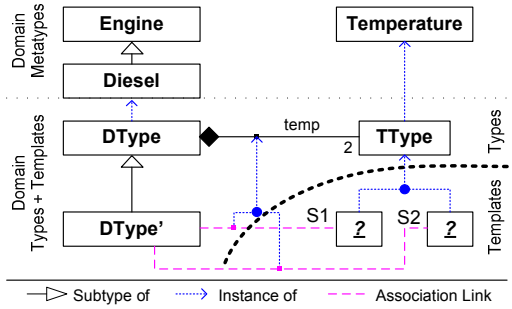

Figure 12. Pre-allocating connector links

Although DType' is an instance of the *type* Diesel, we can use it together with TType instances. While the connector "temp" declares that DType requires two TType sensors, DType' can now already contain *template* sensor instances to make statements about its sensors. The identifiers S1 and S2 are used to distinguish these instances. We call declarations such as S1 and S2 template instances since they do not represent actual, i.e. physically identifiable entities. They can not have a serial number or other identifying aspects. Templates used with types are just used to make statements about the actual instances that will be used when the type is instantiated. One implicit statement is the type, such as TType for S1 and S2. Additionally, template instances can be used to specify constraints such as that S1 and S2 have to measure the same physical phenomenon. Note that the auxiliary type DType' that defines template instances is not necessary; as shortcut, S1 and S2 can directly be specified by DType.

4) Substituting connector links.

Types at higher modeling levels need to be instantiated to eventually yield instances that represent real world entities, as shown in figure 11. This has already been demonstrated for fields and connectors, and now we describe how instantiation works for template instances. Figure 13 shows how an instance of the DType is created by substituting the template sensors S1 and S2 with concrete sensor instances. At the domain instance level, all template instances defined for a type at higher levels must be substituted with concrete, i.e. physically identifiable instances. In the example, the template instances labeled as S1 and S2 must be substituted.

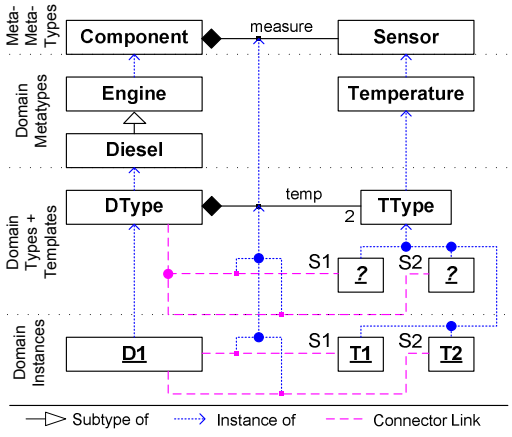

Figure 13. Substituting connector links

D1 is an instance of the specific engine type DType; as such the connector "temp" defined at the domain type level is active at the domain instance level. Concrete sensor instances that can be used to substitute the template sensor instances must also be instances of the sensor type TType. T1 and T2 are therefore possible candidates for substitution.

Each type definition implies an interface that clients may rely on: model elements instantiated or specialized from a certain type must exhibit a compatible interface. Part of that interface is the set of declared fields, but also the set of declared template instances. In our example, since the type DType declares two sensors labeled S1 and S2, the instance D1 must also come with two compatible

sensors with the same labels S1 and S2. Any changes made to the template sensors, such as assigned field values, or established connector links, are part of the interface that clients may rely on. These changes must therefore also be applied to the concrete sensor instances T1 and T2 used by D1. For reasons of clarity the two relationships between the templates and the sensor instances T1 and T2 are not explicitly shown in figure 13.

D. *Gradual instantiation*

With clabjects as introduced in section II.B, a model element can influence model elements at lower levels by defining fields or specifying field values. The idea of *deep instantiation* [10] extends this mechanism, where each field is assigned an integer value called *potency*. With each instantiation, the potency value is reduced by 1. When the potency value reaches 0, a concrete value has to be specified. A potency value of 0 thus means that the field element is part of a model element's instance facet. Assigned potency values therefore implicitly affect the depth of the classification hierarchy.

By definition, a sensor in our domain observes a physical phenomenon. To convert the sensor's measurable electrical signal to the corresponding unit, a conversion function is used. The top level Sensor type therefore can already introduce the two corresponding fields Unit and Conversion. The Sensor type can also define the potency value of the Unit field, as shown in figure 14.

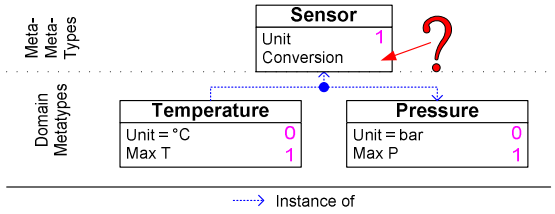

Figure 14. Deep instantiation with potency

The Sensor type, however can *not* define the potency of the Conversion field: its value may be either type specific or instance specific. Thus for our purpose potency values are not applicable. We need a more flexible mechanism that allows specifying field values at any level. Figure 15 shows an example.

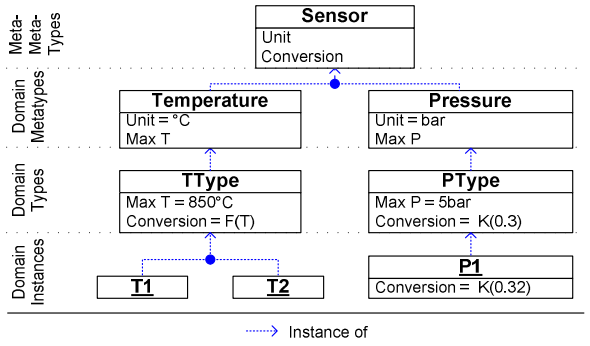

Figure 15. Gradual instantiation

For typical temperature sensor types the conversion function can be precisely approximated by a polynomial function. Instances of this type do not need an individual conversion function. The conversion function would therefore be defined *fixed* for the type. A similar type specific and accurate conversion function can usually not be defined for pressure sensor types, since they have a big variance due to their construction. The type thus can only provide *default values*, but individual sensors need to be calibrated. The instances therefore must be able to override the default value.

In figure 15 we did not distinguish between fixed values, such as the conversion of TType, and default values, such as the conversion of PType. In order to explicitly express the difference, we introduce a "fixed" flag that prevents further changes to fields and connector links. Thus clients, such as the generator for automation system parameters, can rely on this information. Figure 16 shows the example. A lock symbol next to the field indicates that the field's fixed flag is set.

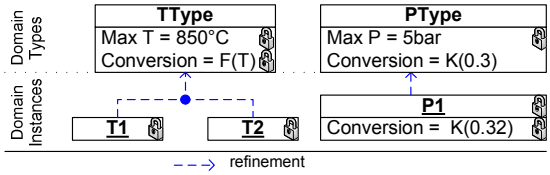

Figure 16. Deep instantiation with fixed flag

In the example TType declares its fields as fixed. Therefore, instances can not change these values anymore. In contrast, PType declares one of its fields as not fixed, so instances can have custom values.

IV. Related Work

Modeling software intensive systems has got a lot attention in the recent years, and standard languages such as UML and SysML have established [2]. SysML, basically an extended subset of UML, provides notions for modeling software intensive systems. However, it is documented in literature that languages based on MOF [12] do not support multi-level modeling [13, 9].

Atkinson and Kühne's original clabject concept [3, 10] primarily focuses on fields and deep instantiation. By and large, we adopted the concept and the handling of fields during instantiation. Deep instantiation, as shown in section III.D, is too strict for our needs and thus a more flexible mechanism is needed, which we call "gradual instantiation". "Ambiguous classification", i.e. a situation in which the type of an instance is not uniquely determinable, is an artifact of shallow instantiation [10]. For connectors, Atkinson and Kühne solve this issue by using deep instantiation. Connector inheritance however is not explicitly discussed in the original clabject papers. In a recent publication, Gutheil et al. [5] elaborate on the issue of how to consistently represent the type and the instance facets of both, clabjects and connectors. The authors argue why UML's internal representation and rendering principles are insufficient and capture their concept in three core principles. We adopted this notation throughout this paper. In addition, we propose to extend connectors with template instantiation to meet the requirements of our subject domain. In general, the clabject notion as discussed in literature provides a sound theoretical foundation, but to the best of our knowledge no real-world case study in an industrial context has been provided so far.

The idea of unifying classes and objects has a long tradition in the object-oriented programming language community, namely in prototype-based languages such as SELF [14]. A SELF-objects consists of named slots that can carry values, which in turn are references to other objects. A special kind of slots, so-called "parent"-slots, refer to parent-objects that are used by the delegation mechanism to dynamically search for an implementation of a message not processed by the child object. SELF uses only one relationship, the "inherits from"-relationship [14, table 1] that unifies instantiation and specialization. Since SELF is a programming language, however, it does not have an explicit notion of associations, which are essential for modeling languages.

Various metamodeling environments are available, such as the Generic Modeling Environment [15], the Eclipse Modeling Framework [16], or the commercial solution MetaEdit+ [17]. These environments have in common that they allow the definition of domain specific languages and provide tools to create editors and generators for code or other artifacts. Although built for different purposes, their meta-metamodels are comparable to a certain degree [18]. Gutheil et al. point out that these tools can operate within an implicit multi-level framework by building a tool chain: The created models are translated into another technology space, where their instances are supported [5]. Nevertheless, none of the mentioned environments directly supports multi-level modeling with clabjects.

V. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated the need for an appropriate modeling formalism to support model-driven engineering in the domain of testbed automation systems. By examining the examples of how to model an engine and its attached sensors, we identified problems that appear in the technical representation of conventional modeling languages such as UML. By employing a clabject-based approach, we avoid these problems. The clabject approach as documented in literature, however, is not sufficient for our needs. So we had to extend the concept with connector template instantiation and gradual instantiation.

To evaluate the concepts presented in this paper we are building a multi-level modeling tool featuring clabjects. In close cooperation with an industry partner real-world testbed examples are modeled. First results already indicate that clabjects and our extensions are well-suited modeling concepts, but we still have to show that they are also beneficial for the transformation system. Nevertheless, this work demonstrates how to apply clabjects to an industrial case of model-driven engineering.

ACKNOWLEDGMENT

The authors would like to thank Stefan Preuer for his contribution to the modeling kernel implementation and Josef Templ for his valuable comments on an early draft.

REFERENCES

[1] Object Management Group, Unified Modeling Language Infrastructure, version 2.1.2, 2007.

[2] Object Management Group, OMG Systems Modeling Language Specification version 1.0, 2007.

[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, "Model-Driven Development: A Metamodeling Foundation". IEEE Software, Vol. 20, No. 5, pp. 36-41, 2003.

[5] M. Gutheil, B. Kennel, and C. Atkinson, "A Systematic Approach to Connectors in a Multi-level Modeling Environment", Proceedings MoDELS'08, LNCS vol. 5301, Springer-Verlag, 2008, pp. 843–857.

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

[7] Object Management Group, Unified Modeling Language Superstructure, version 2.1.2, 2007.

[8] M. Abadi and L. Cardelli, A Theory of Objects, Monographs in Computer Science, Second Edition, Springer-Verlag, New York, 1998.

[9] R. Gitzel, I. Ott, and M. Schader, "Ontological Extension to the MOF Metamodel as a Basis for Code Generation", Comp. J., vol. 50, no. 1, 2007, 93–115.

[10] C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling", Proceedings of UML, LNCS vol. 2185, 2001, pp. 19–33.

[11] S. Friedenthal, R. Steiner, and A. C. Moore, Practical Guide to SysML: The Systems Modeling Language, Elsevier Science, 2008.

[12] Object Management Group, Meta Object Facility (MOF) 2.0 Core Specification, 2004.

[13] C. Gonzalez-Perez and B. Henderson-Sellers, "Modelling software development methodologies: A conceptual foundation", J. Syst. Softw., vol. 80, no. 11, 2007, pp. 1778–1796.

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

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

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

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

[18] Kern H., "Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF", The 8th OOPSLA workshop on domain-specific modeling, OOPSLA Companion '08, ACM, New York, 2008