A Modeling Language's Evolution Driven by Tight Interaction between Academia and Industry

Thomas Aschauer, Gerd Dauenhauer, Wolfgang Pree
University of Salzburg, C. Doppler Laboratory *Embedded Software Systems*
Jakob-Haringer-Str. 2
5020 Salzburg, Austria

firstname.lastname@cs.uni-salzburg.at

Domain specific languages play an important role in model-driven engineering of software-intensive industrial systems. A rich body of knowledge exists on the development of languages, modeling environments, and transformation systems. The understanding of architectural choices for combining these parts into a feasible solution, however, is not particularly deep. We report on an endeavor in the realm of a technology transfer process from academia to industry, where we encountered unexpected influences of the architecture on the modeling language. By examining the evolution of our language and its programming interface, we show that these influences mainly stemmed from practical considerations; for identifying these early on, tight interaction between our research lab and the industrial partner was key. In addition, we share insights into the practice of cooperating with industry by presenting essential lessons we learned.

Categories and Subject Descriptors
D.2.11 [Software Architectures]: Domain-specific architectures – *domain specific languages, model driven engineering.*

General Terms
Design, Experimentation, Languages.

Keywords
Academic-Industry Cooperation, Domains Specific Language, Model Driven Engineering Architecture, Clabjects.

1. INTRODUCTION
Model-driven engineering (MDE) promises a substantial gain in productivity and quality for building large software-intensive systems. In many development endeavors, MDE is applied to cope with the increasing complexity and size of software intensive systems. Domain specific modeling (DSM) can further enhance this gain by aligning the modeling language closely with the problem domain, enabling domain experts to express their solutions in terms of well-known concepts. A rich body of literature is available for the various aspects of DSM, in particular, the modeling language, the modeling environment, and the transformation or generation technology; see for example [1, 2]. Nevertheless, knowledge about the role of the architecture that puts these parts together into a working system is not particularly deep. For example, the fact that concepts of the modeling language usually dictate the architectural choices for building the modeling environment seems reasonable at first glance. Moreover, when using an off-the-shelf language workbench, most architectural choices usually have already been made for you. But what if architectural forces of the modeling environment influence the way the modeling language is constructed or its programming interface is designed, in order to get a feasible solution?

In this paper we report on a DSM endeavor that was conducted in close cooperation between our research lab and an industrial partner over the last three years. The overall goal of this technology-transfer collaboration is to build a MDE-based product for the configuration of automation systems in the domain of engine test facilities. In this realm we developed a domain specific language (DSL), a modeling environment, and a transformation engine. Initially, the development seemed to be quite straight-forward and the first research prototype successfully demonstrated the applicability of the DSL. Since the goal is to build a production-ready system, we needed to develop the prototype further. In this process we encountered that the chosen metamodeling framework did not meet the requirements of industrial reality. Together with other shortcomings, this finding resulted in a complete restart of the project, and it influenced the choice for a new metamodeling framework and its programming interface.

In order to identify these issues and to show how we solved them technically, we examine the evolution of our modeling language in its historical context. We will see that the evolution was not only driven by the domain's mere complexity, but also by the interaction with other parts of the MDE architecture and practical considerations such as maintainability and understandability. In this respect, the close cooperation between academia and industry was invaluable. Nevertheless, this form of cooperation also has its pitfalls, in particular regarding interpersonal and communication factors. We want to emphasize this fact by presenting some important lessons we had to learn.

2. THE HISTORICAL PERSPECTIVE
Initially, our research group set out to improve the state-of-practice of configuring automation systems in the domain of engine test facilities, which are used to run experiments on combustion engines and to collect and record measured data. These facilities are complex systems of systems comprising various "machinery, instrumentation, and support services, housed in a building adapted or built for its purpose" [3]. The test facility automation

ICSE '10, May 2-8, 2010, Cape Town, South Africa

system plays an important role since it is responsible for safe conduction of an experiment by monitoring operation parameters in real-time, for controlling the defined sequence of test steps, and for managing the acquisition of data generated by various sensors and complex instruments. The functioning of the real-time automation system is tightly coupled with the actual setup of the physical test facility, since it has to ensure that all sensors, instruments, actuators, and facility services act safely in concert. Because of the dependence on the actual system setup, the automation system needs to be highly customizable and flexible for supporting a wide range of test facilities. This flexibility, however, comes at the cost of a great effort that is required to customize an automation system for a particular test facility setup. Without adequate software support, customization can be an error-prone and time-consuming activity.

Model-driven engineering was chosen to address these issues. The basic idea is to build a model of the test facility, comprising both hardware and software. This model is then used to derive the corresponding configuration data for the automation system. Our goal was not to invent yet another MDE approach or to build yet another modeling tool, but to select adequate ideas and technologies, ideally proven in practice, and put them to work in a real product.

2.1 Cooperation Setup

There are various ways in which academic and industrial institutions can team up, ranging from mere sponsorship models, to internship programs, to close collaboration. The process of technology transfer in general is divided into separate steps, and usually different parties are involved in these steps [4, 5]. Rombach and Achatz, for example, propose the following technology transfer model: (1) basic research, (2) applied research, (3) prototype in vitro, (4) prototype in vivo, (5) industrial replication, and finally (6) roll out [5].

We use the model above to describe where our work fits in: In our case the collaboration was planned to incorporate steps 2 to 4. Step 2 had to be carried out primarily at our research lab with frequent participation of the industry partner. For step 3 both parties had to work in close cooperation, and step 4 needs to be performed by the industrial partner with scientific support. So the cooperation was a close one. After one and a half years, the company even founded a separate, dedicated branch at our research site to immediately incorporate research results in the development of its product. In retrospect, this was a critical factor for being able to judge the maturity and ramifications of our solutions. More than once the devil was in the details, which are easily overlooked in a prototype implementation in a mere academic setting. As we will see in the following, the evolution of the modeling language nicely exemplifies this observation.

2.1.1 Research Institution

Organizationally our research lab is part of the University of Salzburg. It is partially funded by the *Christian Doppler Forschungsgesellschaft* (CDG), a semi-governmental non-profit organization with the mission of easing access for Austrian companies to leading-edge research [6]. On this account the CDG funds *application-oriented basic research* to bridge the gap between industry and academia by enabling knowledge and technology transfer.

2.2 Phase One

First of all the research team had to gain some knowledge about the problem domain of testing combustion engines. Several workshops and a short internship proved valuable for the researchers to get a grip on the concepts and peculiarities in that domain. We continued with an in-depth requirements analysis to clarify the long-term vision of the product to be built, and to understand the problems to be solved. The initial set of requirements was ambiguous, contradictory, vague, and by no means complete. As we reported elsewhere [7] in detail, the requirements analysis was supported by paper-prototyping and by a mock-up prototype demonstrating essential application scenarios.

It became clear quite early that is was not possible to just fix the old system used for customizing the automation system; it did not provide adequate abstractions, but burdened the domain experts to understand low-level implementation decisions of the automation systems software to be able to customize these systems correctly. Instead, we would provide a domain specific language (DSL), and we also got a good understanding what the abstraction level should be so that the language would enable domain experts to model test facilities rapidly and concisely.

Nevertheless, the choice for an underlying metamodeling framework for the DSL was rather unclear. Prevalent metamodeling approaches based on the Meta Object Facility [8] or similar meta-metamodeling languages didn't seem to match the expectations since they were too inflexible. For example, the domain experts kept telling us that "in the usual case" an engine model would need to comprise its inertia as well as its maximum speed. In some situations, for example when modeling a unique engine, custom properties would also have to be supported, e.g. a correction factor for friction calculation. The language would need to support both cases since "that's real life". According to these requirements we proposed to build the modeling language on a flexible framework that was inspired by prototype-based programming languages, in particular SELF [9, 10]. In essence, a prototype-based language is a special kind of an object-based language that is not based on the usual class-object-dichotomy but instead allows creating a new object by cloning another object that acts as prototype. Afterwards the clone can be modified individually [11]. Figure 1 shows an example. *Engine1* is modeled as an individual object, and *Engine2* is created by cloning *Engine1* and modifying the property values afterwards. The individual property *FrictionFact* for the friction correction factor is also added to *Engine2*.

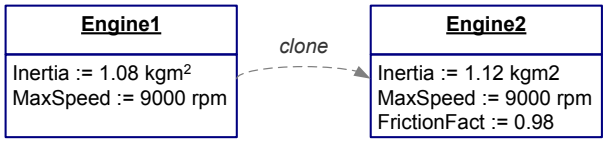

Figure 1. Prototype-based engine modeling.

The idea of harnessing the concepts of prototype-based modeling was appealing not only because of their simplicity. Prospective users were already accustomed to a copy&paste-style. In their daily work, they used previous automation system configuration data to create the configuration for new test facilities. So supporting the same metaphor in our language seemed just intuitive. Furthermore, we had an explicit relationship between each prototype and its clones, so we could trace back every element to its origin.

Consequently many of the disadvantages of using copy&paste, such as lack of change propagation, could be avoided [12].

2.3 Phase Two

In the second phase we built a prototype implementation of the modeling language to test (a) the feasibility of the metamodeling framework, and (b) to evaluate the concepts of the DSL. For the validation an expert of the domain modeled a scenario based on a real test facility. Supporting reuse of model elements to assemble big models efficiently is an essential requirement for the language, since we expect the models to be large, that is, several tens of thousands of elements. We chose to support reuse by means of model libraries populated by users. Library elements serve as prototypes and their clones can be customized according to the rules of prototype inheritance [11].

We further extended the scenario to demonstrate the whole process from modeling to configuration data generation to final execution of the automation system. The results were presented to the company's general management, which in response decided that our approach should be an integral part of their future product portfolio. That decision also triggered the foundation of a separate development branch at the location of our research lab. The demonstration showed that the overall approach was feasible and that the DSL concepts, by and large, provided the right level of abstraction and granularity. At that point we may have claimed the project to be a great success, and that the research prototype has well served its purpose. Nevertheless, a closer look at the system revealed two major flaws right at its core.

2.3.1 Performance

Due to the proof-of-concept character of the implementation, we decided to implement the prototype-based inheritance scheme by means of *copying* [11]. In essence, this means that reusing model elements to assemble larger models resulted in full duplication of data. While for the intended purpose as prototype environment this was acceptable, modeling the real-world scenario was soon rendered impossible due to the excessive memory consumption and associated performance issues.

2.3.2 Modeling Categorization

As it turned out, a simple, unrestricted cloning approach provided too much freedom for our metamodeling framework. In particular, the metamodeling framework did not support modeling *categorization* well. For the generation of the automation system configuration we need to model the properties of each individual engine. Since engines typically share most of their properties, we can reuse the corresponding transformation rules by introducing the concept of an *Engine*, and applying the transformation rules to each clone. Furthermore, engines typically are members of a whole engine series, and all members of a certain series share some characteristics such as their maximum speed, so we can again reuse the corresponding transformation rules. In other words, the engine family is a categorization of engines. In figure 2 we see how this might be modeled with prototypes: First we model *Engine1* as an individual object without any specific values for the attributes. Second, we create the clone *SeriesS1* and specify the value for maximum speed, which is the same for all engines of the series. Furthermore, we add a new property *Tuning* for the specific tuning of the engine. In the third step, we create a clone for a specific engine *myS1Engine* and provide the missing attribute values; we also add an individual attribute with a value, the *FrictionFact* attribute.

A transformation rule for *SeriesS1* can, for example, rely on the fact that *MaxSpeed* is set to 9,000 revolutions per minute to optimize the generated configuration parameters. The potential for optimizations is even larger when the transformation rule relies on structural properties such as the number of cylinders of a *SeriesS1*-engine. This rule, however, cannot be reused safely for clones of *SeriesS1*. Due to the semantics of our prototype-based language, *myS1Engine* can be modified individually. So it is possible to change the maximum speed value, or to add an additional cylinder. Applying the transformation rule thus would lead to failure. Relying on the structural similarity between a prototype and its clones is not possible in our prototype-based language.

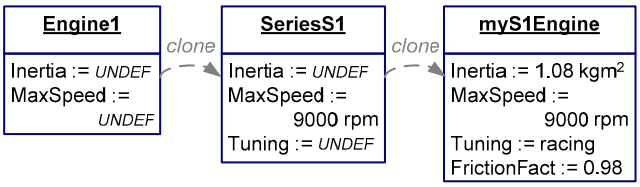

Figure 2. Prototype-based engine modeling.

In the end, due to this degree of freedom in the language, we would either have had to put much more effort into writing more robust transformation rules that do not rely on the inheritance of structure, or taking additional measures in the modeling environment to ensure that clones adhere to the structure of their prototypes. Interestingly, this is also an observation made by the designers of SELF: "Functionality provided at the language level in class-based systems rose to the programming environment level in Self". For the design of a new language one of them "…might be tempted to include inheritance of structure in the language, although it would still be based on prototypes" [10, p. 38].

So the question was which metamodeling framework to use that explicitly supports categorization. Powertypes [13] are a widely used pattern to model such relationships, and are, for instance, supported by the UML [14]. Figure 3 shows how our example could be modeled with powertypes.

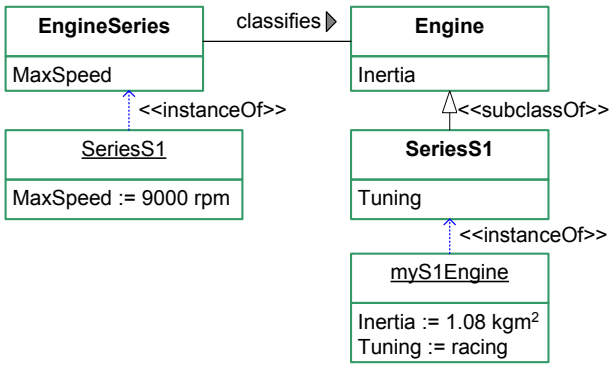

Figure 3. Engine categorization with powertypes.

As we can see in figure 3, the concept of an engine and the concept of an engine series are both represented explicitly in the model and are connected by a special association expressing that an engine series is a categorization of engines. The semantics of this association is that any instance of *EngineSeries* is, at the same time, also a subclass of *Engine*. This is expressed by the modeling

of the entity *SeriesS1* that is represented twice in the figure, i.e. in both roles. The concrete engine instance *myS1Engine* is also shown in the figure. Although this solution supports categorization, we can not represent an individual attribute such as the friction correction factor as a property of *myS1Engine*, because instances in UML cannot contain new structural properties. Furthermore, as elaborated by Atkinson and Kühne, this solution has other disadvantages such as accidental complexity of the resulting models [15]: some concepts might not be part of the inherent conceptualization of our domain, but they are necessary just because they are required for the powertype mechanism.

2.4 Phase Three

Although the prototype-based solution was, at least from our perspective as researchers, reasonable and intuitive, tests of real examples done by domain experts were necessary to reveal its shortcomings for large models. Having identified the flaws, we reconsidered the original design decisions and validated them against the original requirements. We all, including the company representatives, now had a much better understanding of these requirements and their implications. In a one-week workshop we identified some fundamental misunderstandings between the researchers and the company representatives. We were surprised of these misunderstandings, but in retrospect we think that not enough care was taken to gain a common understanding of the terminology. At the same workshop we also developed a set of postulates a better system would have to be built on. Overall, most of these matched the previously built system quite well, e.g. the overall vision, the domain specific language concepts, the user interface, and the interaction patterns between the automation system and the modeling environment. The metamodeling framework however, i.e. our prototype-based inheritance scheme, no longer seemed to be appropriate. The new system would have to support inheritance of structure.

It was due to the determination of the company to get the most suitable solution that they decided to start over again with the same team. Understandably, this decision was quite controversial, if we consider that an effort of *almost 20 person-years* had been invested until then. Later on it turned out that this decision was a key factor to a successful turn-around. The alternative plan was to fix the memory and performance problems with some smart technique and to put more effort in the development of the user interface and the transformation engine to overcome the limitations of the language.

To start the DSL development again from scratch, we had to find an alternative metamodeling framework that better supported the needs of the domain experts. It was not until we have implemented the basic mechanisms based on the workshop postulates that we found that the notion of a *Clabject* was much closer to what we needed. This concept has been proposed by Atkinson in 1997 [16] and since then been studied and extended in several contributions, e.g. [15, 17]. A clabject in principle is a dual-faceted modeling entity that unifies classes and objects; it has a so-called type facet as well as an instance facet. So a clabject can be an instance of some clabject, and at the same time it can be the type of another clabject. Figure 4 shows our engine example modeled with this concept. Note that each modeling entity in the figure is a clabject. As UML does not support the notion of clabjects, we use UML's class symbol for that purpose.

The *Engine* on top represents the most abstract concept and only defines structural properties such as the fields (these are the clabject's equivalent of attributes in UML) *MaxSpeed* and *Inertia*. *Engine* is then instantiated by *SeriesS1*, which represents the engine concept for a particular engine series. It provides a value for the maximum speed field, which means that all engines of this series share the same value, which cannot be changed individually by its instances. Furthermore the series-specific field *Tuning* is introduced. The concrete engine, *myS1Engine*, is an engine of that series and specifies the corresponding field values. Introducing the individual field *FrictionFact* is not supported by the original clabject concept. However, this is one of the extensions that we proposed for the applicability in our domain [12, 18].

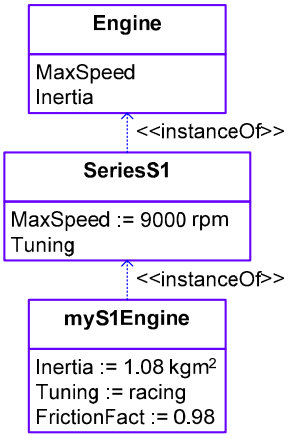

Figure 4. Engine model with clabjects.

After the reimplementation of the language core, now inspired by clabjects to resolve the modeling issues, and employing some memory-saving optimizations to overcome the performance issues, the functionality of the previous prototype was migrated and performance tests assured that the new foundation was solid. Modeling exercises by domain experts and experiments with the transformation engine also showed that the new metamodeling framework was suitable. We also demonstrated that our memory-efficient representation method and the corresponding traversal algorithms scale to large models [19].

2.5 A Test Facility Model in Numbers

Domain experts are currently working on the model of a real test facility to demonstrate the complete cycle of model-driven parameter generation. During configuration of a test facility, the domain expert assembles the model from instances of pre-defined model elements maintained in a library and connects them according to the system to be described. Instances of library elements are customized either by changing their field values or by adding new fields. Based on the assembly of the model elements and their connections, the transformation engine generates the corresponding automation system configuration. When custom fields are added to a library element, the corresponding transformation rule also has to be adapted.

Table 1 shows some quantitative information on a current model covering about 60% of the overall facility. According to the domain experts the hardware part of the model is almost finished; the remaining work primarily is to finish modeling the associated functionality performed by the automation system software and the model transformation rules. For each category we see the number of model elements and the number of fields. For the fields we also include how many were added or modified,

since these numbers indicate to which degree pre-defined library elements need to be customized. The visualization of model elements in diagrams is also stored in the model, so the corresponding numbers are included as a separate category.

Table 1. Size of a test facility model (~60% completed)

Category	Elements	Fields		
		Total	Added	Modified
Hardware	9120	26625	0	1548
Software	3692	18764	28	17
Input/Output	3422	11006	0	231
Visualization	10440	71148	0	2829
Total	26674	127543	28	4625

3. CODE EVOLUTION

After examining the overall project development, we will now look at the evolution of the source code. To this end, we mined our source code repository for quantitative data. First of all we extracted monthly snapshots of the active development branch. On each of these snapshots we calculated two metrics: the number of classes and interfaces on the one hand, and the number of code lines on the other hand. To see how the different parts of our solution evolved, we categorized the software modules, called "assemblies" in .Net terminology [20], manually into the following categories: *BASE* denoting the metamodeling framework, *DSL* denoting the domain specific language, *GUI* denoting the user interface, and *TSA* denoting the transformation engine and the transformation rules. The category *TOTAL* summarizes the numbers for all categories including modules not captured otherwise, such as import/export functions. Because in the first phase most of the created artifacts were informal documents and test programs, we did not include this phase in our measurements.

Figure 5 shows the graphs of the number of classes and interfaces of our system. The TOTAL graph reveals some interesting facts. In the beginning (*i*) we can observe the trend of continuous growth as we would expect. Perhaps the most striking feature can be observed at the end of the second phase at instant (*ii*), where all software development was stopped. With the beginning of phase 3 we started from scratch. Note that the codebase we had by then was not completely abandoned, but was further used for some time by domain experts to experiment with the DSL concepts and by new team members that needed to acquaint themselves with the project. Since the development of this codebase was a dead end, we did not include it in our metrics that represents the active development branch only.

After the restart we can observe a steep growth of classes (iii). This can be explained by the fact that a considerable amount of code could be migrated from the earlier version to the new one. Later on the number of classes steadily grows, except for a short negative spike that represents a cleanup of the DSL (iv), where we got rid of obsolete language elements that were introduced now and then, but that later became obsolete. It is important to note that most of the classes in the *DSL* category were automatically generated. In fact, the language elements of the DSL are modeled as classes in a UML model with a custom profile. The UML model is then used to generate C# classes that represent the language elements in our implementation. If we consider the trend of the *DSL* category, we can see that after the initial startup and except for the cleanup, the corresponding code was quite stable. As a matter of fact, the increased number of classes between instants (ii) and (iii) is only due to the new representation based on clabjects. The number of DSL elements is the same at both instants, and we used the same UML model to generate the corresponding classes. The continuing steep growth of TOTAL classes after instant (iv) can be explained due to additional developers that joined the team and primarily worked on the user interface and on the transformation engine.

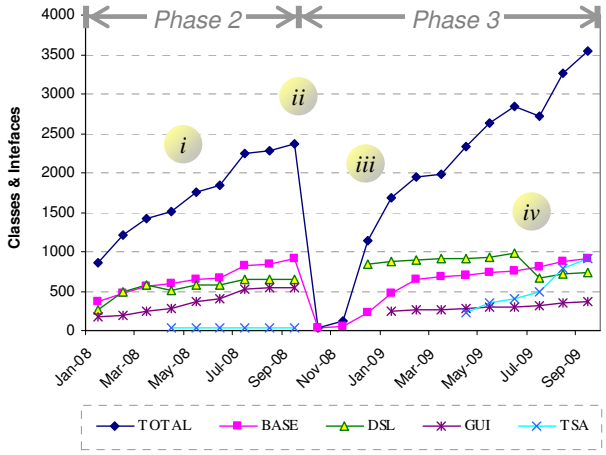

Figure 5. Classes and interfaces in active development branch. *Significant sections are (i) steady growth, (ii) phase 3 as a complete restart, (iii) steep growth due to migration of features from phase 2, and (iv) DSL cleanup.*

In figure 6 we see the same period as in figure 5, but this time we measured lines of code (LOC). To fully understand the absolute numbers, it is important to note that we only counted lines that were actually compiled into .NET Common Intermediate Language instructions [21]. An empty method body or class definition, for example, does not count in this LOC-metric, since no instructions are generated.

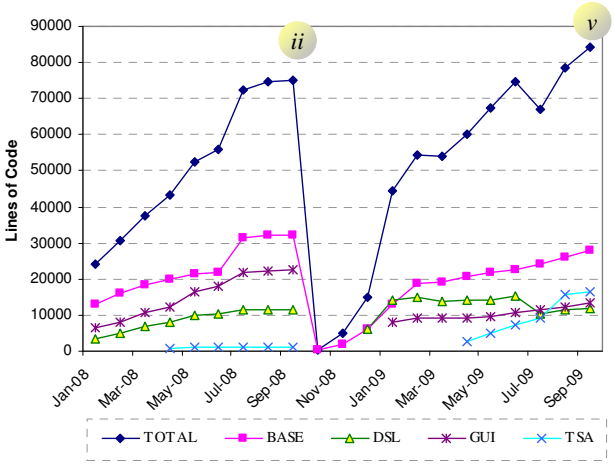

Figure 6. Lines of code in active development branch. *The graph by and large resembles figure 5. Interestingly, the LOC-counts at (ii) and (v) differ only slightly, whereas we have quite different class-counts at the same instants in figure 5.*

3.1 Discussion

By and large, the class graphs of figure 5 and the LOC-graphs of figure 6 look alike. However, a remarkable feature is revealed when we observe that for the total amount of LOC in figure 6 the growth from instant (*ii*) to instant (*v*) is about 12%. Comparing the number of classes for the same instants, we see that the growth there is significantly higher, about 49%. Since at the later instant the software provided more functionality than at the earlier instant, we might reason that now we can do more with the same amount of code; it is, however, structured differently and in smaller classes. Another explanation along the same lines is that because agile techniques such as refactoring were applied [22], the code was steadily enhanced and cleaned up so that its verbosity decreased. In other words, the code evolved into smaller, self-explaining methods and classes.

Considering the most remarkable feature of both graphs, i.e. the complete rewrite of the metamodeling framework at the beginning of phase 3, one is tempted to ask: What would have happened if this endeavor was conducted by an industrial company without academic involvement? Would they have stopped the project after the language flaw and the memory problem were revealed? Would they have just fixed the performance issues, and continued? Or similarly: How would the project have proceeded in a purely academic setting? Would we have got to the end of phase 3 and revealed the maintenance flaws, or would we have declared the project as "successfully finished" even before we got there? Well, there certainly are no definite answers to these questions. But the graphs indicate that, arguably, the close cooperation between industry and academia was an important factor for the endeavor's development.

3.2 Limitations

The results of our analysis presented in this section certainly are biased by the data that has been used, and so care must be taken when attempting to generalize them. The threats to validity that appear to be most relevant are: (a) we have used the data of one system only, which is focused on a certain market niche in the domain of automotive test systems, and (b) the metrics used represent, by their very nature, only specific aspects of the system under study.

4. LANGUAGE INTERFACE

In phase 3 we have resolved the language issue and the memory problems. Over time the system grew and the industry partner implemented most of the additional code contributing to functionality, user interface, and usability. For a broader accessibility of the modeling environment several convenience features such as a more mature graphical representation and some import-tools were needed. Since the resources at our site were limited, development and test activities were transferred to a near-shore center in Europe and also to an off-shore center in India; this explains some of the steep growth of code we saw in figure 5 after instants (iii) and (iv).

At the end of the third phase the results were again presented to the company's general management. We happily reported the success story of our new approach. Nevertheless, with more and more code being added to the system by more and more developers, we saw complexity creeping in and felt that in the long run maintainability will become an issue.

4.1 Generic API

As one of the root causes of the maintainability problem we identified that the low-level application programming interface (API) to access model elements was not well-suited for the code written by the increasing number of developers. In our current implementation, the interface to access model elements is generic. It basically contains methods to retrieve model elements, to navigate along connectors (these are the equivalent of associations in clabject terminology) and along the inheritance graph, and to access fields and their values. This way the modeling environment's core is independent of the concrete DSL since it only relies on these meta-concepts. Functionality such as persistency and the query engine are also implemented generically through this API.

Our modeling environment is built around a compact core which comprises the metamodeling framework, the DSL, a generic visualization framework, and services such as persistency. In order to support specific application scenarios the core environment is extended by plug-in modules, which can depend on certain DSL-concepts. Our visualization plug-in, for instance, provides specific renderings for DSL elements representing hardware such as sensors or actuators. Another plug-in realizes an importer for a fieldbus device description language. Although these modules depend on certain DSL concepts and must make assumptions about their structure, they still have to use the low-level API to access model elements. The following code exemplifies the kind of programs developers have to write[1]:

```
public void SetMaxTemp(IModel model) {
  IElement sensor =
    (from element in model.TopLevelElements
     where element.Type.Name == "TempSensor"
     select element).First();
  IField max = sensor.GetField("RangeMax");
  max.Value = 420;
}
```

The code snippet searches the first temperature sensor in the model and assigns the *RangeMax* field the value 420. In some situations it might be quite convenient to be able to access an element's fields via a generic API; here, however, it causes tedious programming. Moreover, the access through names such as "TempSensor" or "RangeMax" might cause maintenance problems since their validity cannot be checked statically but only at runtime.

Interestingly, we find similar customization issues in other DSL environments, such as MetaEdit+ [23] or the Generic Modeling Environment (GME) [24]. Using these environments or similar ones usually 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. The visualization of a DSL concept usually can be customized, e.g. by providing individual icons or more sophisticated by creating of a custom plug-in. Creating plug-ins to customize these environments, be it either for visualization pur-

[1] The actual interface to access model elements is more sophisticated, but we omit all technical details for reasons of brevity.

poses, for import/export functionality, or for model transformations, means that one has to use the generic API. Similarly to the example we have seen above, this leads to lengthy code due to the low abstraction level of these interfaces. For some often-needed customizations these tools provide special interfaces. MetaEdit+, for example, provides its own scripting language called MERL for developing model transformations that allows the usage of DSL-elements directly in the language [2]. In the general case, however, one still has to stick to the generic API.

4.2 Domain Specific API

A domain specific interface for accessing model elements supports a high-level style of programming much better. So in our language we provide two interfaces: the low-level API as described above, and a second, domain specific API that is automatically created from the DSL model. Figure 7 shows the approach: Plug-ins can access model elements either through the *Generic API*, or through the *Domain Specific API*.

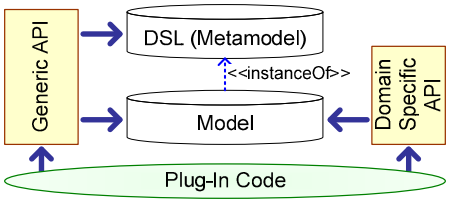

Figure 7. Generic and domain specific API for model access.

When using the generic API, a plug-in might as well access elements of the *DSL*—since our language is based on clabjects, DSL-concepts are not different from other model elements. Interestingly, this is an immediate benefit of the so-called *Orthogonal Classification Architecture* as described by Atkinson and Kühne [17]. Its basic idea is that in any clabject-based system we have two orthogonal classification dimensions: In one dimension we have the classification between the programming language classes, such as *IElement* or *IField*, and the modeling language elements, such as *Sensor* or *RangeMax*, as instances of them. This dimension is called *linguistic classification*. The second dimension, called *ontological classification*, is the classification between the modeling elements at different abstraction levels, e.g. between the DSL element *Sensor* and test facility model elements such as *TempSensor*. All elements in the ontological classification dimension are at the same time also classified by linguistic elements, so we can treat them uniformly—in our case, we can use the generic API to access both, model elements and DSL elements.

The ultimate goal of the domain specific API is giving the developer the ability to write code in terms of higher abstractions than with the low-level access. The code for our former example of assigning a maximum range value to the first sensor in the model should rather look like accessing a sensor object similar to common object-oriented languages, for example as follows:

```
public void SetMaxTemp(IDslModel model) {
  Sensor sensor =
    (from element in model.TopLevelElements
     where element is TempSensor
     select element as Sensor).First();
  sensor.MaxRange = 420;
}
```

To evaluate whether realizing the domain specific API by generating the corresponding classes *a priori* is feasible, we need to take a closer look at how models represented by the *Model*-layer in figure 7 are built. In our case this is not only one user-built model, but it is a layered set of libraries and user-models that define concepts at different abstraction levels. Figure 8 shows a typical example: The *Domain Concept Library* defines concepts that are well-known in the domain, such as an *engine*, a *measurement device,* or an *electrical connector*; these elements are not specific to a certain manufacturer. This library is provided by our partner company. The *Manufacturer Library* is also provided by our partner or by a third-party vendor; it contains models of specific products such as a measurement device for emissions analysis or a specific kind of engine. This library builds on the concepts defined in the domain concept library. Even a customer can prepare its own library of customized versions of the third-party vendor's measurement devices or engines. Finally, the *Test Facility Model* represents the facility for which we generate the configuration parameters.

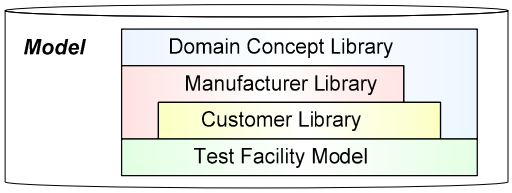

Figure 8. Example of libraries comprising the model level.

In our customizable modeling environment, plug-in modules can depend on specific libraries. The vendor of measurement devices might, for example, provide a module that can update model information of devices by importing a proprietary configuration file. An engine-manufacturer might provide a plug-in that provides consistency checks for engine models. Since test facilities are individually built systems, it is even possible that a customer's R&D department needs to provide custom check routines or tools. Since we can not foresee the number and kind of such plug-ins, implementing the domain specific API by means of generating classes a priori would mean to generate classes for each of these model layers.

4.2.1 Implementation

Generating the domain specific API would, in the end, mean to generate tens of thousands of classes. So as an alternative we use a *precompiler* for implementing the domain specific API. The precompiler translates the code into a representation that accesses the low-level API only, as shown in figure 9.

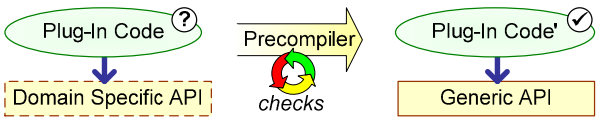

Figure 9. A precompiler for realizing the domain specific API.

The main advantage of this solution is that we do not need to generate any classes a priori, and so we can also access library elements such as *TempSensor* as classifiers in the domain specific API. In addition, we can implement domain specific checks—a feature that justifies the term *precompiler*, as opposed to a *preprocessor* performing only macro expansion. We can, for example, check whether the model library really contains the element

TempSensor and whether it is derived from the DSL element *Sensor*; so all its instances are guaranteed have the property *MaxRange*. It is also possible to enhance the syntax for domain specific purposes. In our case, for example, we plan to provide a specific syntax for specifying units such as °C for the temperature sensor's maximum value. Furthermore, as a future extension a domain specific query language is planned, so the precompiler will be used as a framework for seamless integration of both languages.

Providing a specific API in domain specific modeling environments is not a new idea. Nordstrom et al. [25], for example, propose a technique for automatically generating a domain specific API for metamodels built with the GME. The basic idea is to automatically generate an API for each element of the DSL. They describe in detail how the generic modeling concepts of GME can be encapsulated by automatically generated C++ wrapper classes. For C# a similar implementation also exists [26]. Since these approaches use a generative approach and because GME is based on a fixed metamodel hierarchy, the corresponding domain specific API is based on metamodel elements only. In contrast, due to the utilization of a precompiler and as a consequence of the Orthogonal Classification Architecture, our approach also allows to use library elements as classifiers in program code.

4.2.2 Expected Impacts on Code Base

We are currently finishing the implementation of the precompiler. In the short term we expect that this affects the code base only insignificantly, in contrast to the effect we saw when switching from prototypes to clabjects. The transition now is expected to be rather smooth since the domain specific API is only an extension to the generic API; we can refactor the code bit by bit. In the long term, however, we expect that the lines of code will grow much slower than they do now. But more importantly, we expect that the complexity of the code decreases significantly, which is an important aspect for maintainability and understandability.

5. LESSONS LEARNED

During the evolution of solution we gained some insights in the practice of transferring technology from an academic institution to industrial practice. Remarkably, the essential lessons are not about technical details, but about organization of the collaboration, about social factors, and about communication.

Embark on projects that allow for a fundamental change. Many projects in which industry and academia collaborate require the almost impossible: fix an existing system but do not change it too much. Narrow space for developments, however, does not encourage fundamental innovations.

In our case we set the conditions so that we were free to fundamentally rethink how the configuration system should be designed. Within about half a year it was clear that the effort needed to overcome the enormous complexity, which was accidentally added to the existing system over the years, was not justifiable. Accidental complexity means that not the complexity of the domain caused the overly complex software, but the short-sighted fixing of various problems and probably some bad design decisions [27]. No matter what caused the unmanageable complexity, the existing system could not just be overhauled. Rewriting the complete control system also was impracticable since the system has a good reputation, it has a long-standing leadership role in the market, and the effort required was simply too much for the resources we had available. So a happy medium—based on the existing control system, but radically different from the current configuration approach—had to be found, and these constraints finally broke the ground for applying model-driven engineering.

Be careful with terminology and your own bias. Failing to establish a well-defined terminology that is understood by both parties can cause serious communication problems: you use the same term but mean different things. This issue, often encountered in requirements engineering, is also called a *conflict* in *terminological interference* [28].

In our early workshops, for example, the term "component framework" was used by both parties. Different interpretations of that term caused a fundamental misunderstanding of the problems that the academic institution was supposed to solve. The software scientists meant a *technical framework* for components, which are units of composition with contractually specified interfaces and explicit context dependencies. The domain experts, in contrast, meant *domain components* that describe test facility elements such as an engine or a speed controller; these components can be assembled to describe a complete test facility. While these two interpretations of the term "component" are somehow related, they require different problems to be solved. A technical component framework needs to take care of e.g. component deployment, registration and discovery, process boundaries, etc. A domain component framework, in contrast, needs to provide means for describing the domain entities concisely and mechanisms for assembling these descriptions in order to compose sub-system descriptions. Surprisingly, although we were quite aware of the potential pitfall of an inconsistent terminology in the requirements elicitation phase, it took us months to clarify that issue. In retrospect, we think that a too loose usage of technical terms and our own bias were the root causes for that misunderstanding.

Solve fundamental problems without deadlines and perform major rework when necessary. When the industry partner needs results at some deadline, one is tempted to go for a quick fix instead of a solid solution. It is important to make clear that such quick fixes easily introduce accidental complexity again.

At the end of the second phase we had to present some milestone results, and so the whole team was occupied for weeks in preparing the software to be able to present the required functionality. Due to the tight schedule and other pressure sources more and more quick fixes were added to the system. For the presentation this was fine. On the code, however, this had a bad impact, so that in the end everybody was unhappy with its quality—we fell into the same trap of tentative implementations that caused so many problems in the old system. With the project restart we had a chance to do better the second time—not only with better concepts, but also with an improved discipline. It is not that we didn't have tight schedules later on, but now the system's quality had top priority, and the key stakeholders seemed to understand that thorough deliberation leads to better solutions.

Find an advocate. The industry partner needs to understand that there is a different working culture at the research institute. We think that this is not a problem if the cooperation is a loose one, where a regular meeting every few weeks is sufficient. But in a close cooperation, where work results from one side are immediately used by the other side, conflicting interests quickly arise. It takes an experienced leader to shield the team from too much

pressure. Particularly in long running projects this requires stamina, patience, and conviction. Experience reports of other DSL projects also describe the necessity of having such a person, called an *advocate* by Wile [29]. We also made the experience that it is indispensible that the advocate is strongly backed by the management when times get rough.

In our case we were lucky to have an experienced project manager on the industry side who saw the potential of different but complementing working cultures. He has formerly led research and development projects within the company, but also with academic cooperations. Our advocate had to be backed by the upper management to be able to shield the team from other groups in the company that would have pushed for a conventional project management, that declared the project as failed when the performance problems were revealed, and that would have liked the project to be discontinued when budgets had to be cut due to the global economic downturn.

Expect emotions and be aware of miscommunication. As Ramos et al. point out, the introduction of radically new software and the vision of a changed work reality is never free of emotions [30]. Underestimating or not taking seriously these inevitable emotions and beliefs can endanger the system's acceptance, and thus the success of the whole endeavor.

After about six months our team presented the prototype, which was enthusiastically received by the industry partner's management. We captured the demonstrations as video sequences and, with the best of intentions, distributed them inside the company. Some people apparently took this the wrong way: since this was the first reliable source of information on our project, the videos caused disturbance in the current system's development team. We were told not to distribute information in this format any more. In the following weeks our advocate was quite busy to shield the project team from very emotional discussions on the meaningfulness of the project. It took months of careful communication to convince the key persons that the new system does not only serve some academic hypothesis, but also eases their daily businesses.

Displacement vs. integration—have both in due course. On the one hand, displacement from the partner company's site encourages the development of new ideas because one is less exposed to the tacit assumptions that domain experts usually share. Moreover, having a *beginner's mind*, that is, being an *ignoramus* with respect to the problem domain, enables to think outside the box [31]. This is an advantage of loose research partnerships. On the other hand, one runs the risk of acceptance and integration issues if, for example, one does not strictly adhere to the company's development process. For a long-running and close cooperation as ours, we think that having both at the right time helps to create innovative solution with good acceptance.

In the early phases we were geographically and organizationally displaced from the company's headquarter. For the development of our concepts this was useful since we could ask all our "stupid questions" in a protective environment. Nevertheless, it was also important to have easy access to domain experts in that phase, an observation that confirms the experience of others [31]. When our prototype reached a more mature stage it was valuable that the company founded a separate branch at the research site: until the integration of the development team in the industry-partner's process, some key people still regarded our project as just an academic prototype that could not stand up to

just an academic prototype that could not stand up to real-world problems.

People transfer eases technology transfer. When the company opened the local branch, some developers stayed at the academic institution and continued to work on research questions, and others transferred to the company to advance the software to product-quality.

In our experience this transfer eased the cooperation in later phases. The obvious advantage for the company was that these developers built the initial system architecture and so there was continuity in the development team, and they were part of their organization. The advantage for the researches was that theirs roles changed from core development to more conceptual responsibilities. Moreover, the communication and the appreciation of each other's contribution improved; we think that social factors played an essential role here.

Be proud of applied research. Not only practitioners have mistaken our work, we had similar experiences in the academic world. Due to the lack of generalizations, which we could not draw in the early stages of our endeavor, our work was dismissed because it appeared as an application of "existing concepts" only, or as a "nice engineering exercise" with low scientific value. Developing the domain specific modeling language and getting it ready for production certainly was a lot of work. Perhaps the same effort could have generated more publications if invested in other topics.

In the end, however, we could show that model driven engineering is feasible for our target domain. We could further demonstrate advantages and weaknesses of modeling languages based on prototypes and clabjects in an industrial context. Many of the subtle details did not occur until we further developed the research prototype into production-quality. So we completely concur with David L. Parnas' statement in an ICSE plenary talk: "If we want our ideas to catch on, we have to put them into products. There is a legitimate, honorable and important place for researchers who don't invent new ideas but, instead, apply, demonstrate, and evaluate old ones" [32].

6. CONCLUSION

In this paper we presented the evolution of a domain specific modeling language for engine test facilities. By examining the history of the endeavor we saw that although the domain abstractions were quite stable from the beginning, our initial choice for a metamodeling framework was not ideal and thus led to a complete project restart. But even with an appropriate framework at hand, the reality of a flexible architecture and a distributed multinational development team had influence on how this framework was used. Consequently, we had to provide a domain specific API to raise the level of abstraction used by developers.

What we can learn from these observations is that the mechanisms we need in a metamodeling framework are not only influenced by the necessity to model a domain concisely, but also by the context in which the framework is applied. For a practical MDE solution dependencies between the architecture and its constituting parts arise; instead of encountering them on accident, they should be examined more systematically from the beginning.

Another conclusion is that the cooperation between the academic and the industrial world is not always easy—there are

many pitfalls in the technology transfer process. As indicated by the lessons learned, the essential factors in a close cooperation are of social and communicational nature; these are usually difficult to control. Nevertheless, it is rewarding to develop an idea into a mature product in this setup: it can raise new potentials for the partner company, and it can bring new insights for the research partner. We hope to help others who set out for a similar journey to avoid the one or other of the inevitable pitfalls.

7. ACKNOWLEDGMENTS

We thank our industry partner, the AVL List GmbH, and in particular the members of their research & development team.

8. REFERENCES

[1] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316-344, 2005.

[2] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[3] A. J. Martyr and M. A. Plint. *Engine Testing: Theory and Practice*. BUTTERWORTH HEINEMANN, 3rd edition, June 2007.

[4] S.L. Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2-3): 111-124, 1999.

[5] D. Rombach and R. Achatz. Research Collaborations between Academia and Industry. In *29th International Conference on Software Engineering (ICSE'07), Workshop on the Future of Software Engineering (FOSE'07)*. IEEE Computer Society Press, 2007.

[6] Christian Doppler Forschungsgesellschaft. http://www.cdg.ac.at

[7] T. Aschauer, G. Dauenhauer, P. Derler, W. Pree, and C. Steindl. Could an agile requirements analysis be automated? - Lessons learned from the successful overhauling of an industrial automation system. In *Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs: 14th Monterey Workshop 2007, Revised Selected Papers*. LNCS 5320, pages 25-42. Springer-Verlag, 2008.

[8] Object Management Group. *Meta Object Facility (MOF) Core Specification*, Version 2.0, January 2006. Document formal/06-01-01.

[9] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 227-242, New York, NY, USA, 1987. ACM. Also published as *SIGPLAN Notices*, 22(12):227-242, 1987.

[10] D. Ungar and R. B. Smith. Self. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1-50. ACM, 2007.

[11] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438-479, 1996.

[12] T. Aschauer, G. Dauenhauer, and W. Pree. Towards a generic architecture for multi-level modeling. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009)*, pages 121-130. IEEE Computer Society, September 2009.

[13] J. J. Odell. Power types. *Journal of Object-Oriented Programming*, 7(2):8-12, May 1994.

[14] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, Nov. 2007. OMG document formal/2007-11-04.

[15] C. Atkinson and T. Küuhne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345-359, August 2008.

[16] C. Atkinson. Meta-modeling for distributed object environments. In *IEEE International Enterprise Distributed Object Computing Conference*, pages 90-101. IEEE Computer Society, Oct 1997.

[17] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36-41, 2003.

[18] T. Aschauer, G. Dauenhauer, and W. Pree. Multi-level modeling for industrial automation systems. In *35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2009)*, pages 490-496. IEEE Computer Society, August 2009.

[19] T. Aschauer, G. Dauenhauer, and W. Pree. Representation and traversal of large clabject models. In *12th International Conference On Model Driven Engineering Languages And Systems (MODELS 2009)*, LNCS 5795, pages 17-31. Springer-Verlag, 2009.

[20] Ecma International, Geneva, Switzerland. *Standard ECMA-335: C# Language Specification*, 4th edition, 2006. Also approved as ISO/IEC 23271:2006.

[21] Ecma International, Geneva, Switzerland. *Standard ECMA-334 : Common Language Infrastructure (CLI)*, 4th edition, June 2006. Also approved as ISO/IEC 23270:2006.

[22] R. C. Martin and M. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall PTR, 2006.

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

[24] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44-51, 2001.

[25] S. Nordstrom, S. Shetty, K. G. Chhokra, J. Sprinkle, B. Eames, and A. Ledeczi. Anemic: automatic interface enabler for model integrated computing. In *2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, pages 138-150. Springer-Verlag, 2003.

[26] T. Vajk, R. Kereskényi, T. Levendovszky, and Á. Lédeczi. Raising the abstraction of domain-specific model translator development. In *16th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 31-37. IEEE Computer Society, 2009.

[27] F. Brooks, Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10-19, 1987.

[28] N. Niu and S. Easterbrook. So, you think you know others' goals? A repertory grid study. *IEEE Software*, 24(2):53-61, 2007.

[29] D. Wile. Lessons learned from real dsl experiments. *Science of Computer Programming*, 51(3):265-290, 2004.

[30] I. Ramos, D. M. Berry, and J. A. Carvalho. Requirements engineering for organizational transformation. *Information & Software Technology*, 47(7):479-495, 2005.

[31] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179-184, 1995.

[32] D. L. Parnas. Software aging. In *16th International Conference on Software Engineering (ICSE '94)*, pages 279-287. IEEE Computer Society Press, 1994. (Invited plenary talk).