Domain-specific Modeling as an Enabling Technology for Small and Medium-sized Enterprises

Donald Barkowski
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2238
donald.barkowski@iese.fraunhofer.de

Thomas Kuhn
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2177
thomas.kuhn@iese.fraunhofer.de

Christian Schäfer
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2121
christian.schaefer@iese.fraunhofer.de

Mario Trapp
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2272
mario.trapp@iese.fraunhofer.de

ABSTRACT

In this paper, we present experiences and the outcomes of applying domain-specific modeling techniques to the domain of diagnostic systems within an industry project. The purpose of the project was to develop a diagnostic system that is easy to configure to different facilities without a deeper knowledge of software development. Although the project was relatively small, we will show that the use of domain-specific technologies proved to be of value, was superior to traditional approaches, and turned out to be a key enabling technology in the project.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE)

General Terms

Design, Languages

Keywords

DSL, Experience report, Industry, Configuration language

1. INTRODUCTION

In this paper, we present the outcomes of applying domain-specific modeling techniques to the domain of diagnostic systems. Experiences are taken from an industrial cooperation project with a small enterprise with about 20 employees. The company is not a software development company; rather, its main business model is trading in hydraulic components and providing hydraulic solutions with a special focus on small batch series. It was not an evaluation project with industrial participation, but it was the project objective to develop a concrete product prototype providing the basis for our customer to enter a new market segment.

One of the services offered is the condition monitoring of hydraulic systems. This means that different properties like pressure, viscosity, or pollution are continuously measured and interpreted in order to detect and predict anomalies. Since each and every hydraulic system is unique, different properties are of importance, different sensors must be used, and different anomalies must be defined. The condition monitoring system (CMS) must therefore be tailored to each facility.

While the hardware of the CMS can be a generic one and therefore may remain fixed, its firmware must be very flexible. A traditional approach towards the programming of such a diagnostic system would thus mean manually adapting its firmware (probably in C) for each single environment. The obvious disadvantages of such an approach include high development efforts, a long time to market, and probable quality problems. In the given context, this approach was not an option at all, since the company is focused on the development of hydraulic systems and has no sufficiently skilled software developers. Being a small-size enterprise, employing software experts was not an option, either. For many small or medium-sized enterprises that are not focused on software development, the situation of having no sufficiently skilled software developers is certainly familiar.

To overcome these problems, it was the objective of the project to develop a configuration software enabling the engineers to graphically configure the CMS. Considering the hints and patterns for when to develop a domain-specific language, as they are described for example in [1], we quickly came up with a domain-specific solution approach comprising the definition of domain-specific languages along with the provision of a code generation framework. This relieves developers of the diagnostic system of the task of hand coding the CMS firmware. Instead, all software development know-how that is necessary for developing a CMS is encapsulated in the language and the code generators. The development process for new systems thus becomes fast and cost-efficient, as it is only a matter of a few clicks to adapt the CMS to new environments. Moreover, due to the automatic code generation, the results are less error-prone and therefore need less testing effort.

In this paper, we describe the background of our DSL based solution for configuration of condition monitoring systems and point out our experiences with introducing a DSL into an industrial setting. The remainder of this paper is organized as follows. First, we give a brief summary of the main objectives of the CMS. Thereafter, we present the basic concepts of our solution, which are based on domain-specific modeling principles [4], and prove their applicability to real-world industrial problems. Details about the implementation and the underlying tool chain, which is based on Eclipse, follow next. Finally, in the conclusion, we briefly summarize the benefits of our approach and point out how domain-specific modeling technologies serve as a technical enabler for small and medium enterprises (SMEs).

2. PROBLEM DESCRIPTION

The condition monitoring system (CMS) is intended to support the monitoring and diagnosis of primarily hydraulic facilities, although it may be extended to almost arbitrary facilities. Its main objectives can be split up into basically four subtasks. First of all, the system must collect measurement data from the facility such as pressure, viscosity, or the like. Therefore, it has to be connected to new or already available sensors integrated into the facility. However, as the output of sensors may vary due to many different influencing factors, the CMS must be configured with a characteristic curve for each connected sensor. This curve maps the values of a sensor's output signal (i.e., current or voltage) to the respective value of its measured quantity (e.g., pressure).

Second, the system has to continuously analyze the collected measurement data in order to predict or detect anomalies in the facility's behavior. To this end, it uses basic error detection mechanisms, such as range checks or the comparison of the data values from two different sensors. In case of a detected anomaly, the system must then be able to do some basic intervention by activating/deactivating digital outputs. This makes it possible, for example, to initiate a full or partial emergency shutdown of the facility in a critical situation.

Furthermore, as the system is intended to also support remote diagnostics, the sensor data must be made available at a remote location, too. Therefore, it is necessary for the system to transmit all collected data and all analysis data to a remote database. In addition, it shall provide some notification about the occurrence of anomalies to a dedicated cell phone number via SMS, so that a person not present may be immediately informed about a critical situation at the facility.

Finally, a monitoring software shall be developed that is used to access the database and visualize the facility's data, thus allowing for remote supervision and diagnostics. The monitoring software shall display value patterns of selected sensors and outputs and shall show the occurrence of anomalies. The monitoring software is based on the very same platform as the configuration software. As it therefore provides no additional value to the understanding of the domain-specific modeling concepts used within this paper, the remainder of this paper is focused on the development of the configuration software and we will not regard the monitoring software in further detail.

Apart from the aforementioned tasks, it is of particular importance that the CMS can be fast and easily adapted to different facilities. As the engineers performing this task have almost no software engineering know-how, this particularly means that the system shall be configurable mostly graphically with the possibility of automatically generating the firmware for its microcontroller from the created graphical models. This, in fact, is exactly the description of a domain-specific modeling approach. Consequently, the tool that is used to adapt the CMS to a specific facility is actually a domain-specific development environment.

3. Overall language development approach

For building our domain-specific solution approach, we carefully were adhering to best practices and lessons learned like those described e.g. in [3]. The major building blocks of our solution are depicted in Figure 1. Before we explain the single elements of the solution in more detail in the subsequent chapters, let us regard the overall solution idea as an overview.

In the first step, the engineers define several models describing the CMS. This basically includes the specification of the sensors used by the CMS as well as the definition of the anomalies to be observed. To this end, we provide the engineers with several domain-specific graphical editors, which we developed in Eclipse using the freely available plug-ins EMF (Eclipse Modeling Framework) and GMF (Graphical Modeling Framework). The editors are based on several meta-models describing the terms and concepts used within the given problem domain. Thus, the meta-models define the admissible "language" for the editors, which in turn guarantee that the engineers can only create CMS models that follow its syntax and structure. Hence, the engineers are given some editors that enable them to think and model a CMS in terms of their domain, while relieving them of the burden of having to map their concepts to constructs of a programming language.

The meta-models used within the editors do not contain any details about the underlying hardware structure of the CMS, despite the number of its inputs and outputs. Accordingly, the engineers do not need to worry about these details during development, but instead they define a model of the CMS that is completely independent of its platform (platform-independent model, PIM). Nevertheless, as the platform details are necessary

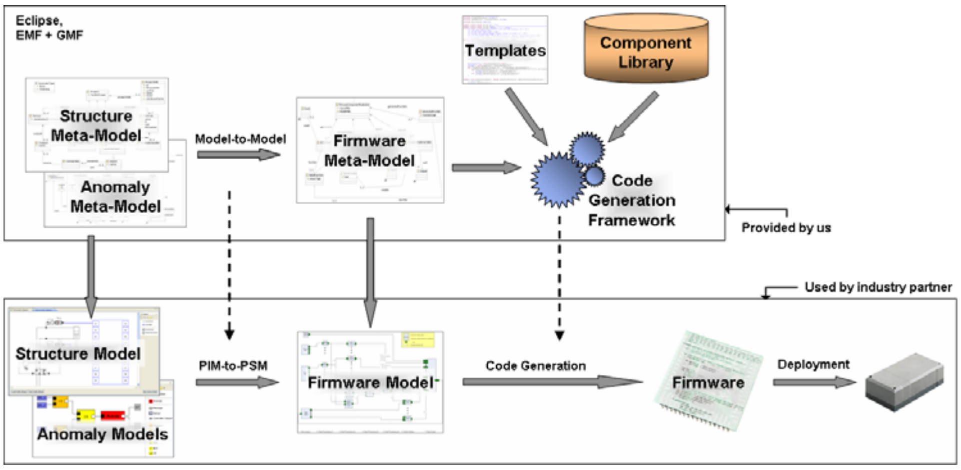

Figure 1: Building blocks of our DSL based solution

during the final code generation, they must be added later on. This is done in the next step, where the PIM is transformed into a platform-specific model (PSM) of the CMS firmware[1]. However, for the engineers, this is just a matter of pressing a single button: The transformation is done fully automatically without any further user interaction. This is possible since the hardware platform is fixed and has been defined in a corresponding meta-model. Thus, all information required is included in the model-to-model transformations.

In the next step, the PSM is used as input for the code generation. As a result of this step, C source code is generated, which is ready to be compiled and deployed to the CMS microcontroller. Again, for the engineers, code generation is only a single click without any need for further manual configuration steps. Instead, all necessary information such as code templates and implementations of standard components is already included in the provided code generation framework. Finally, for the last step consisting of the compilation of the firmware and the flashing of the generated executable to the CMS microcontroller, a standard tool chain (gcc, avrdude) is used.

3.1 Structure modelling

The structure model is used for modeling the overall system and facility layout and for the initial definition of anomalies that may occur in the facility's behavior. In the first step, the engineer must define the sensors that are used by the CMS. Figure 2 shows the meta-model of a *Sensor* definition. Basically, it consists of attributes for indicating the sensor's type (current or voltage output), its measured quantity, and its *characteristic curve*. The characteristic curve itself is a piecewise linear approximation made up of an arbitrary number of *points*. Moreover, each sensor possesses an *Outport*, which is used for connection purposes.

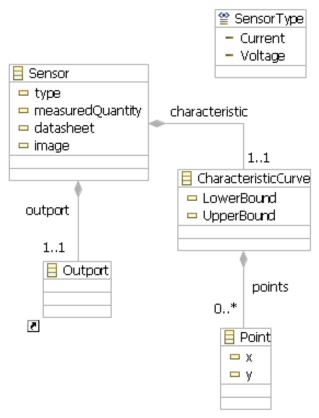

Figure 2: Meta-model of the sensor element

The sensors are realized as reusable components, i.e., if the same sensor type has already been used in a previous project, the sensor definition can be reused. It can be easily adapted to the new facility by modifying its characteristic curve.

[1] The underlying concept for this separation into PIM and PSM is the Model Driven Architecture (MDA) proposed by the OMG [5]

Once the sensors have been defined, the actual CMS structure can be modeled. The underlying meta-model for this task is depicted in Figure 3. The key element in the meta-model is a *Project*. Almost everything that is modeled in the structure model is somehow related to or belongs to a *Project*. First of all, a *Project* has an associated *ProjectInfo*, which is just a container for additional information such as its author, ID, and the name of the project-specific database. Furthermore, a *Project* consists of up to 10 *Sensors* and one *Controller*, which have to be placed into the editing area. In order to increase clarity and facilitate this placing, the *Project* may have a drawing of the facility associated with it. However, this drawing is solely laid out in the editing area's background. Besides that it has no semantic meaning.

The *Controller* represents the microcontroller of the CMS that is responsible for the sensor data acquisition, evaluation, and transmission, and for which the firmware needs to be generated. In our case, the *Controller*'s hardware was fixed to ten *In-* and *Outports* each, with four of the *Inputs* being more precise than the others (with a resolution of 12 rather than 10 bits). The *ControllerInports* may be connected to the *Sensors*' *Outports* as indicated by the *Connection* element. As the amount of data may become rather large depending on the sensors' output rates, the data that is sent to the database by the *Controller* may be reduced individually for each sensor. Therefore, the user specifies a time interval within which minimum, maximum, and average values are computed. Only these three values are then sent to the database, making it possible to significantly reduce the amount of stored data.

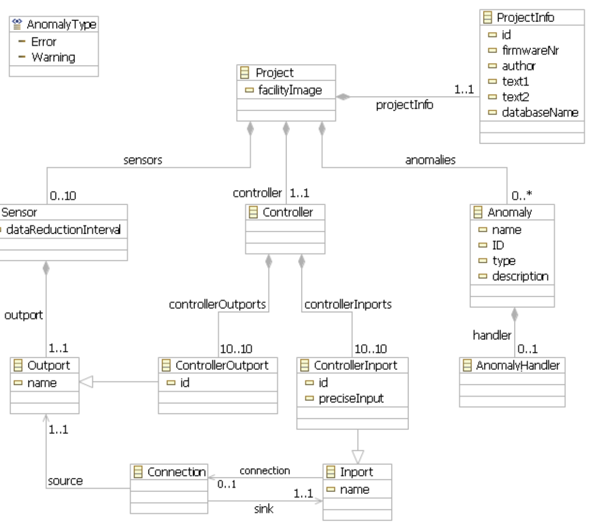

Figure 3: Meta model of the project element

Once the CMS structure has been laid out, the A*nomalies* that may occur in the facility need to be defined. Each *Anomaly* is specified by a name, a unique ID, a type describing its severity (either error or warning), and an optional description, which may be used to provide more detailed information to the user. Moreover, each *Anomaly* may include an *Anomaly Handler*. The *Anomaly Handler* is responsible for detecting the associated anomaly and defining a reaction in case of its occurrence. This behavior is defined in the anomaly model presented in the next section.

3.2 Anomaly modelling

The anomaly model is used for specifying a single anomaly handler, i.e., each anomaly handler is defined by a separate anomaly model. The process of anomaly detection and reaction may be divided into five subsequent tasks as depicted in Figure 4.

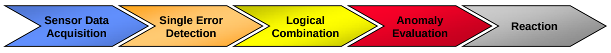

Figure 4: Process of anomaly detection and reaction

In the first step, the data from the sensors that are relevant to the detection of an anomaly must be acquired. Therefore, the relevant sensors are selected and added to the anomaly model.

Next, the available sensor data needs to be evaluated. To this end, three basic error detection mechanisms (cf. Figure 5) have been defined. These mechanisms take the signals from one or two sensors as input and output a Boolean value indicating whether the respective error has occurred. The first error detection mechanism is a *MinMax* block, which may be used to check adherence of a sensor signal to a certain value range. Accordingly, the range must be specified by a minimum and a maximum value. The second error detection mechanism is a *Gradient* block, which allows for checking a sensor's value pattern to prevent violation of a minimum or maximum gradient. Finally, the engineer may use a *Compare* block in order to compare the data values from two different sensors. Here, the values must be equal within a specified tolerance.

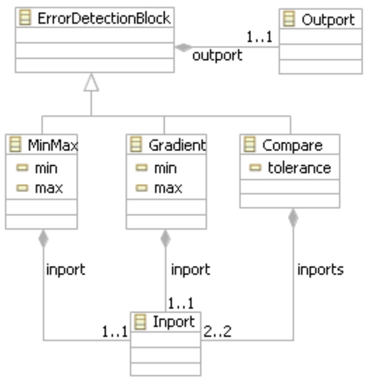

Figure 5: Meta model of the error detection blocks

In order to allow the definition of more complex functions for detecting an anomaly, the use of multiple blocks for error detection is possible. In this case, the results of the single error detection blocks must be combined logically. To this end, three well-known basic logical blocks are available: an *AND*-, an *OR*-, and a *NOT* block.

After the logical combination of the error detection blocks, we have one single Boolean flag basically indicating whether a certain anomaly has occurred or not. This flag may now be further evaluated in order to do some sort of "debouncing". For this purpose, a minimum time period can be specified within which the flag must hold true in order to switch on the anomaly. Accordingly, a minimum time period can be specified within which the flag must hold false in order to switch it off again. Both cases make sure that short outliers will not toggle an anomaly.

In the last step, we need to specify the reaction of the CMS to an anomaly. This basically means that we need to determine which of the controller's outputs become activated and whether an SMS message needs to be sent.

3.3 Model transformation & code generation

In order to generate code, additional information about the target platform is required. Hence, the PIM is first transformed into a platform-specific model (PSM) of the firmware, which is then used for generating C source code. The PSM basically is a component-based model of the CMS firmware with a dataflow execution semantic between its components. Except for sharing the same CPU, the components operate independently of each other, i.e., all hardware resources and interrupts are used exclusively by one component at any one time. As its name implies, the PSM is designed exclusively for usage on a given target, in our case the 8-Bit microcontroller of the CMS. This holds true for both the meta-model of the PSM and a concrete model. Hence, it cannot be used on other platforms without rework. This not only affects the firmware model as a whole, but also its single components, each of which comes with preliminary C implementations and templates that are used as building blocks during code generation. Consequently, as most of these implementations and templates are tailored for usage on the target CPU and many of them directly access hardware resources, they cannot be reused on other platforms without modifications.

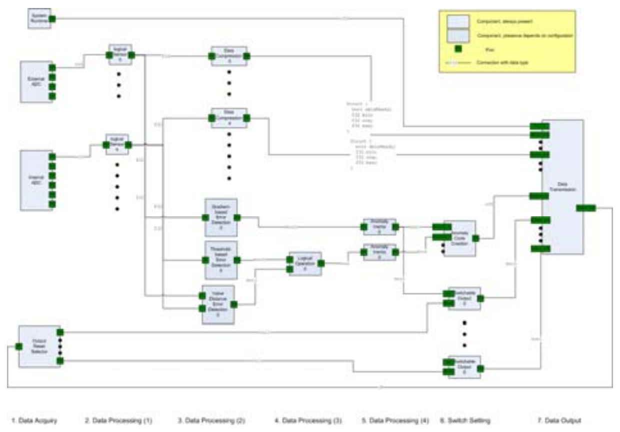

Figure 6: The PSM of the firmware

The basic structure of the PSM is depicted in Figure 6. The dataflow proceeds from left to right. While many parts of the PSM structure are independent of the PIM and thus are included in every concrete PSM generated by an engineer, other parts are not. For example, the depicted components *System runtime* and *Data transmission* as well as the connection between them are included in every concrete PSM. Contrary to this, the components *Error Detection*, *Logical Operation*, and *Anomaly Inertia* are dependent on the concrete anomaly models defined in the underlying PIM. Here, each anomaly model of a PIM is transformed into at least one *Error Detection* component, an optional *Logical Operation* component, and exactly one *Anomaly*

Inertia component. Nevertheless, model transformation is straightforward and no further user interaction is required.

After the PIM has been transformed, the PSM is used as input for code generation. To do so, the PSM's components have to be tailored according to the customizations defined by the engineer in the PIM. Accordingly, the components' code generation templates can be configured using different parameters. For example, the *Data Compression* components calculate a minimum, average, and maximum value for their respective channel (i.e., Sensor). Therefore, they need to know the user-specified interval length of the appropriate sensor, which thus has to be passed as a parameter to the component's template. Despite such simple parameters, more complex parameterizations are needed, too. For example, the *Logical Sensor* components need to know about the sensors' characteristic curve in order to compute the mapping of the input signal to the measured quantity. Such complex parameterizations are realized by generating header files that define C symbols for these parameters. The symbols, in turn, are built from the settings in the structure and anomaly models.

One problem during code generation is the collision of variable names. These collisions are inevitable in any case where two instances of the same component are used (e.g., the *Logical Sensor* component). However, the solution to this problem is relatively simple and just makes use of define statements that rename every globally visible variable or function in every component such that they have unique names.

Another aspect that needs to be dealt with is the scheduling of the components. This is realized statically, which is possible since the PSM is a dataflow model with a runtime-invariant structure. The execution is cycle-based and each cycle is divided into seven scheduling phases (cf. bottom of Figure 6). In each cycle, all components are executed from left to right according to the phases. The communication of the components between the scheduling phases is realized by copying data from the component outputs to the inputs of connected components. A cycle is initiated every 5ms by the scheduler in order to maintain a required sampling rate of 200Hz for the Analog-Digital-Converters (ADC).

Finally, during code generation, appropriate makefiles are created, too, allowing for the build process and the flashing onto the microcontroller to be automated. In this way, any programming language details are completely hidden to the user.

3.4 Technical Details

We implemented the meta-models and their respective editors in Java as plug-ins for Eclipse. Eclipse is primarily known as a software development platform for Java, but one of its key features is that it is easily extensible through its plug-in mechanism. One can write one's own tools that seamlessly integrate into the existing platform, making use of already existing plug-ins. For the editors of the CMS, we based their implementation on the freely available plug-ins EMF (Eclipse Modeling Framework, [6]) and GMF (Graphical Modeling Framework [2]).

EMF provides modeling and code generation frameworks that allow developers to define a model (e.g., a UML model or any other XMI- (XML Metadata Interchange) serialized model) and generate Java code for this model. Although the Java classes generated in this way are only code representations of the model and do not have any particular functionality defined yet, they may be modified and extended. In case of subsequent changes to the model and regeneration of the code, the modifications are preserved. In our case, the models we defined within EMF were the meta-models of the modeling language for the CMS, i.e., the models within EMF specify the "language" we use for modeling the structure and the anomalies of the CMS.

In addition to the generated Java classes for the CMS modeling language, EMF is capable of generating a plug-in with a set of classes that supports interoperation with other EMF-based tools. Furthermore, it provides the possibility to generate a basic editor. But as this editor provides only a very rudimentary look-and-feel, we additionally used the GMF plug-in to implement more sophisticated graphical editors. GMF itself makes use of the EMF-generated plug-ins and allows for the generation of a graphical editor. One "solely" needs to specify a graphical definition containing the graphical representations for the elements in the EMF model, a tooling definition containing the editor's tools (such as a creation tool for a specific model-element), and a mapping definition that puts all these aforementioned definitions together. Similar to EMF, the GMF-generated editor only provides basic functionalities and in our case it was necessary to further modify the editors in order to suit our needs. Nevertheless, it served as a good starting point and made things much easier than starting from scratch.

4. Experiences and benefits

After project end, it turned out that developing and integrating our solution in Eclipse with EMF/GMF was the right decision. When introducing the system to customers, these were not only interested to buy complete and finished CM systems, but also were interested in the ability to make minor changes to CM-Systems on their own. Selecting Eclipse and EMF/GMF as base technologies kept license costs down for the DSL runtime and enabled the developed technology therefore also for smaller companies, which opened up a new market segment for condition monitoring systems. This outweighed the fact that initial development costs were higher due to the additional coding effort for realizing the prototype.

Compared to competitors that also do offer tailored condition monitoring systems, our partner is now able to deliver tailored software systems faster, and with only a fractions of development costs of a conventionally development monitoring system. This enables customized monitoring systems not only for highly specialized and extremely costly hydraulic systems, but also for mid-ranged systems, which make up an enormous market. While our solution may not be perfectly applicable for the most specialized systems that do require very specialized algorithms, it provides a perfect solution for the large majority of systems.

After introducing the system to potential customers, an overwhelming flood of requests and orders for the system was received. As already mentioned earlier, these customers were not only interested in a pre-tailored CM-system for their installation, which would have been considered a success already, but also in the ability to tailor systems by themselves. This opened up another market segment for our partner, which was not expected when starting the project.

Our own experience with DSL projects in an industrial context is that they tend to be successful, if they solve a clear problem with respect to the needs of their users. It is important that DSLs are simple, intuitive, and that they are solving a real and focused problem. In this case, huge productivity increases are possible, which provide considerable benefits to industry. To our experience, the biggest gain of domain specific languages is the omission of repetitive concepts. For instance, the described DSL focuses on detection algorithms and sensor types, which are changing between installations. Overall system behavior, its state-based behavior, and memory management are common for all systems – domain experts do not need to care about these unnecessary details, which would be very time-consuming when developing CM-Systems the traditional way. Therefore, reduction of unnecessary flexibility is in our opinion the key to success, not only for domain specific languages, but for efficient development of complex software systems in general.

This also holds for modeling languages in the field of software engineering, where currently standardized languages are dominating. These languages are often very broad and flexible but too bloated to be useful for practitioners. Instead, slim and well tailored domain specific languages are necessary that integrate with customer specific constraints, guidelines, and processes. These simple languages tend to be accepted by domain experts, prevent errors, and yield considerably increased efficiency.

5. Conclusion

The presented project adds more evidence that domain-specific modeling technologies are ready to use and work even for industry purposes. This does not only include large companies with big development budgets and additional personnel skilled in software engineering, but it particularly holds true for small and medium-sized enterprises which are able to adapt quickly to new technology. In our case, e.g., the CMS configuration environment, the monitoring software, and the hardware were developed with reasonable effort in about ten man-months, finally making it affordable for our customer. Moreover, in the given domain with the given problem and taking into account the business conditions of our customer, the domain-specific modeling approach seemed to be the only reasonable solution. For our customer, neither employing new software engineers nor training the present staff in software development was possible. As a consequence, a traditional approach, i.e., manually programming and adapting the CMS to different facilities, was not feasible. Thus, the domain-specific modeling approach can be seen as the key technology for enabling a new business model for our customer.

Hence, from our customer's viewpoint, the development of a CMS for a particular facility now has become fast and cost-efficient, while at the same time the CMS is probably of a higher quality. This is due to the new possibility of configuring the CMS by making use of the terms and concepts of the domain, and not needing to map these concepts onto a lower programming language. Now domain experts may define the logic part of the CMS firmware while the code generation framework assures that it is implemented correctly and with equal quality, no matter who modeled the firmware.

From our viewpoint, although the project presented here was relatively small, it gave us some reasonable insights into applying domain-specific modeling technologies for an SME. First of all, we can conclude that the complexity of setting up a domain-specific modeling framework remains manageable. In the given context, for instance, the languages reflecting the concepts of the condition monitoring domain were relatively clear and fixed with close boundaries. Consequently, the meta-models and the corresponding code generators were not subject to many changes. Other domains and projects meeting the same criteria may thus profit from domain-specific technologies as well. On the other hand, if the systems to be developed become more complex and require different concepts from multiple domains, a solution with only a single domain-specific language will not be sufficient. In fact, multiple languages will be needed which may be used in parallel, each of which addressing different aspects of the system. To this end, further research needs to be done to identify and define mechanisms that allow for such a multi-language development. Moreover, it is not likely that domain specific languages replace general purpose modeling languages. In fact, it is more reasonable to tailor such general purpose languages to a given domain by extending and modifying them using domain-specific language concepts. By this means the intuitiveness and expressiveness of domain specific languages is combined with the flexibility and the advantages of standardized tool chains provided by general purpose modeling languages. Particularly in combination with a multi-language development environment, this leads to a very powerful, intuitive, and flexible framework for the development of complex and heterogeneous embedded systems.

6. REFERENCES

[1] M. Mernik, J. Heering and A. Sloane: *When and How to Develop Domain-Specific Languages.* In ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316–344.

[2] R. C. Gronback: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Longman, 2009.

[3] D. Wile: *Lessons learned from real DSL experiments*. In Elsevier: Science of Computer Programming, Vol. 51, No. 3, June 2004, pp. 265-290.

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

[5] Object Management Group, Inc: *Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1*. omg/03-06-01, published in 2003.

[6] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks: *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, part of the Eclipse Series series, 2009.