A Component based Approach for Modeling Expert Knowledge in Engine Testing Automation Systems

Gerd Dauenhauer, Thomas Aschauer, Wolfgang Pree

Technical Report
May 7, 2010

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

A Component based Approach for Modeling Expert Knowledge in Engine Testing Automation Systems

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

***Abstract*-Test automation systems used for developing combustion engines comprise hardware components and software functionality they depend on. Such systems usually perform similar tasks; they comprise similar hardware and execute similar software. Regarding their details, however, literally no two systems are exactly the same. In order to support such variations, the automation system has to be customized accordingly. Without a tools that properly supports both, customization as well as standardization of functionality, customization can be time consuming and error-prone. In this paper we describe a modeling driven approach that is based on components with hard- and software view that allows defining standard functionality for physical hardware. We show how this way most of the automation system's standard functionality can be generated automatically, while still allowing to add custom functionality.**

I. DOMAIN INTRODUCTION

The modeling approach we describe here was developed together with our industry partner, a leading provider of a specific kind of automation systems, mainly used in the automotive industry during development of combustion engines. Such systems by and large comprise similar components, such as (a) the engine to be tested, (b) one or more dynamometers for generating load, (c) a shaft for connecting the engine to test with the dynamometer, (d) devices for providing fuel, water and oil, (e) I/O devices for gathering measurement data and controlling the hardware, and (f) various measurement devices, for example for analyzing exhaust gases. The control PC (g) finally executes the automation system software, controlling the actuators according to a test procedure and recording measurement data. Fig. 1 schematically shows a typical example.

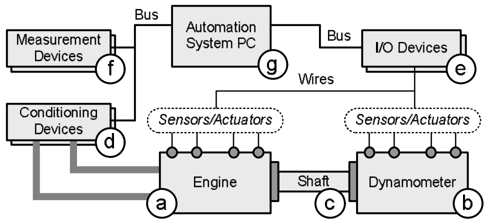


Fig. 1. A typical engine test system. It shows the basic hardware components usually found in a system used during development of combustion engines.

Many aspects of the automation system software need to be configured according to the demands of a test procedure. For example, the parameters of PID controllers for the cooling water supply, the maximum engine speed and the action to be performed if the threshold is exceeded, or the error reaction in case the shaft breaks. A typical test system can have 10,000 configuration parameters with 120,000 values. Most of these parameters are static during a run of the test procedure. Our research was initiated because of our partner's limited tool support for this kind of configuration tasks, which is largely based on configuration files. Parameter values can therefore be checked by the automation system software only during start-up. Getting a system running is thus an error prone process that can take up to weeks, even for experienced personnel.

Our proposal is to raise the level of abstraction from configuration files to models of the test system. In order to sufficiently represent the automation system, we clearly have to model the software functionality executed by the automation system. But we also have to model relevant parts of the hardware. For example, the wiring between I/O devices and sensors is relevant for tracing a signal used in software functions back to its sensor and measurement location. Other physical components such as the shaft or transmission are relevant, since their physical properties have an influence on the software functionality, e.g. the inertias of all rotating components are necessary for the engine speed controller.

Fig. 2 shows how our modeling approach fits into the existing automation system; it indicates that we model both, hard- and software of the automation system, and generate configuration files compatible with the current automation system software. We do not modify the existing system.

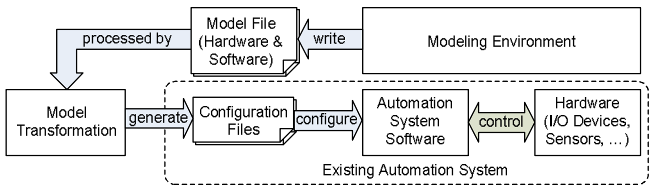

processed by
Model File (Hardware & Software)
write
Modeling Environment
Model Transformation
generate
Configuration Files
configure
Automation System Software
control
Hardware (I/O Devices, Sensors, ...)
Existing Automation System

Fig. 2. Configuration through model driven engineering. Configuration files are not created by hand anymore, but generated from the model file instead.

Although literally no two deployed systems are exactly the same, they usually still have a similar structure: instead of a shaft connecting an engine to the dynamometer directly, there may be a transmission added in between; or instead of one single dynamometer, there may be a differential gear with two dynamometers. Similarly, the functionality executed by the automation system is largely the same for each test system, for example: (1) torque and speed of an engine must be controlled, (2) the shaft must be monitored for breakage, or (3) the cooling water temperature must be controlled.

In order to illustrate these common functionalities and highlight our specific modeling approach, we focus on the shaft between the engine and the dynamometer. Despite its simplicity, the shaft is actually a safety critical component: it has to withstand mechanical forces that can break it. A break can cause severe damage to the system and may injure personnel. The automation system software must therefore continually check that the shaft is intact and shut down the system immediately if a breakage is detected. The basic idea is to monitor the speed at both ends of the shaft. A large difference between the two speeds thus signals a breakage. A small difference, however, must be allowed since, however rigid the shaft seems, it acts as a torsion spring.

Other parts of the test system, such as the dynamometer require much more elaborate functionality and even for the shaft additional functionality can be required. For example, wires in proximity of the shaft are used for additional break detection: if the shaft breaks, it will tear apart some of these wires and interrupt their electrical signals, or it will cause short-cuts among other wires. In order not to get lost in technical details, we thus stick to the simple shaft example throughout the paper. For an introduction to functionality required for engine testing, we refer to Martyr and Plint [1].

A great deal of the software functionality is common among test systems. Obviously, instead of manually defining the functionality each time from scratch, it should be defined by experts only once. Such standard functionality should be stored in a library for reuse in multiple systems. Following our example, the library may contain different break detection functions. If a shaft is used in a test system, the corresponding functions would be added, thus eliminating much of the routine work and reducing the chances of making errors.

II. Existing Modeling Approaches

Since we need to model both, hard- and software of a test system, we first have to take a look at available modeling languages. Generic languages such as the systems modeling language (SysML) [2] allow modeling hard- and software, but its generality is both, its strength and its weakness. For example, our targeted users should not be confronted with unnecessary diagram types such as a *use case diagram* or a *package diagram*. On the other hand, users should be provided domain-specific modeling language concepts, including electrical wiring. While in SysML electrical wiring of a system could be modeled in the *internal block diagram*, the notation does not have built-in semantics e.g. of plug compatibility, and tools could thus not guide users very well.

Modeling environments with a smaller, dedicated scope usually come with more concrete semantics. Functionality executed by the automation system software can be represented by dataflow programming languages such as MATLAB/Simulink [3], and Hardware can be modeled using computer aided design (CAD) systems, such as EPLAN [4] for the electrical wiring. Using multiple modeling languages for hard- and software does not seem to be practical, though. For example, tracing back an input to a software function to the sensor via I/O devices, electrical cables and plugs would involve multiple models from different tools.

In certain cases, models in different languages can be synchronized programmatically, using integration platforms. GeneralStore [5] is one such platform supporting integration of different software specific models, such as structural information in the Unified Modeling Language (UML) [6] and control rules in MATLAB/Simulink through transforming models in specific metamodels to and from a shared, generic metamodel. GeneralStore, however, focuses on embedded electronics and to our knowledge does not support CAD systems. Even if hard- and software models could be integrated, users still had to work with multiple different tools, disrupting the workflow.

Föderal [7] follows similar goals as we but has a wider focus on special purpose machinery. In this context, machines are designed specifically for a customer but still share some similarities. Where traditionally whole CAD drawings or programmable logic controller (PLC) programs were copied from existing projects and adapted to fit the new machine, the new approach is to identify smaller generic building blocks that can be parameterized and assembled to create new machines. Mind8 [8] is a tool that can then be used to generate the CAD drawings or PLC programs from these building blocks. Users finally work on these generated artifacts using the original tools to define custom features not yet anticipated in the building blocks. As with the GeneralStore approach, we think that using independent tools for hardware and software aspects disrupts the workflow.

Modeling hard- and software of a test automation system, however, is not sufficient for our approach. We also need to express dependencies among certain hardware components and corresponding software functionality. In product line engineering (PLE) [9], there is the concept of a feature model which describes choices that can be made for a family of similar products. Each product of the family is represented by a distinct selection of features. Features may not be selected arbitrarily, but they can depend on the selection of other features. The kind of products created with this approach typically comprises source-code (e.g. for creating the software for a range of similar mobile phones) or UML models [10]. The feature model thus defines the scope of a family of similar products.

We could think of applying PLE in our domain, too, for example, by making the break detection a feature that is mandatory if a shaft is used. In contrast to the clearly defined scope of product families in PLE, our domain requires much more flexibility, i.e. we think that it is impossible to define a single feature model describing all possible types of test automation systems with reasonable effort. PLE tools such as DOPLER [11] thus split up one big feature model into multiple smaller parts that they hope are easier to manage.

A major problem with PLE, however, is the insufficient integration of feature modeling and selection with the actual modeling environment: products are usually created only by making selections within the feature models, and the generated product is not manually modified afterwards. If, for example, a model of a test automation system is generated, users must not manually delete certain mandatory model elements. Since feature modeling is done outside the modeling environment, the modeling environment has no notion of a mandatory feature to prevent this, though.

So, in order to provide an appropriate basis for modeling test automation systems, we provide a modeling environment with a notion of physical hardware and software functionality executed by an automation system. We also support declaring parts of the model mandatory in order to represent domain expert's knowledge, e.g. for typical software functionality associated with certain hardware components. The rest of the paper present our own modeling approach and briefly sketches how it fits into the process of generating configuration parameters for a concrete automation system.

III. The Integrated Modeling Approach

In the section I, we showed that engine test systems share many commonalities regarding their hard- and software. Models of such systems can thus benefit from libraries of reusable, predefined template components. We also saw that hardware components such as a shaft often require standard functionality executed by the automation system. In order to further simplify the creation of models and free the users from having to manually associate the hardware and software aspects each time again, our approach introduces the concept of a *component* with *views* for *hardware* aspects and *required software* aspects. When a model of the system's hardware is created from predefined components, the model of the system's software is to a large extent created automatically. This section uses the simple shaft break detection example to illustrate the principle behind our modeling approach.

A. The Hardware View

In our approach, the hardware view is quite straight-forward. Components are represented by rounded rectangles with small boxes representing the ports, such as electrical plugs or flanges. Fig. 3 shows an example of a shaft. These components are used to create models similar to fig. 1.

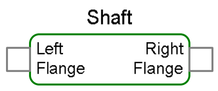


Fig. 3. Shaft hardware view: interface. A shaft is a simple hardware component with flanges at both ends, represented as ports.

The ports of a component define (part of) its *interface*. As the term interface suggests, there can also be an *implementation*, i.e. a component can be composed from sub-components. Such a composition can be represented using additional diagrams. A shaft, however, is a simple component so it does not need to describe further details. Interfaces of other physical components comprising a test system such as the engine and the dynamometer are defined analogously.

Components can intuitively be connected to other components through compatible ports, for example a flange can be connected to exactly one other flange, and an electrical plug can be connected to other electrical plugs. Fig. 4 shows part of a test system, following the example in fig. 1. The interfaces of both, the engine and dynamometer comprise additional ports not shown here, including electrical wires and plugs, e.g. for various sensors or electronic control units, and pipes and fittings, e.g. for fuel supply and exhaust gases.

In order to reduce complexity of the diagrams, the graphical representation can be adjusted by the users to show only part of the component's interface definition, changing the shape, or freely placing its ports.

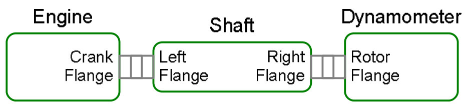


Fig. 4. Part of the test system hardware view: implementation. Engine, shaft and dynamometer are connected through their flanges.

In general, a component such as the shaft in fig. 3 is a self-contained unit that can be managed independently in a library, i.e. its interface does not contain connections to the engine or dynamometer. Such a component can serve as a *template*: copies of the template can be used as sub-components within other components. For example, the shaft in fig. 4 is a copy that just happens to have the same name as the shaft template in the library, i.e. they are distinct entities. Besides copying from a template, components can be created from scratch. We postpone templates until later.

B. The Software View

For describing software functionality associated with a hardware component, we use the dataflow notation similar to MATLAB/Simulink, i.e. boxes usually represent functions and ports represent *input* and *output signals*, and connections between these ports represent the flow of signals. Fig. 5 a) shows an example of the shaft break detection as described in the introduction. The example contains a delta function computing the difference between two speed signals, and a reaction function that triggers an emergency stop of the system in case the difference exceeds a certain limit. The threshold, as well as the kind of reaction are properties of the reaction function; we skip properties in this paper.

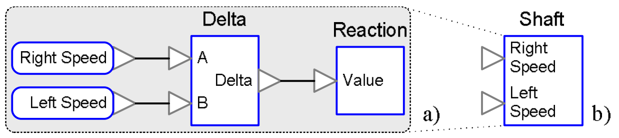


Fig. 5. Break detection software view: implementation a) and interface b). The functionality uses two speeds, as well as a delta and a reaction function.

The rounded blocks are not functions; instead, they represent *required input signals*, analogously to subsystem ports in MATLAB/Simulink that have to be provided by some other source. The break detection model makes no assumptions about their origin. According to fig. 1, in a typical system the shaft directly connects an engine and a dynamometer. Both components usually have a speed sensor attached, so these signals can be used as inputs for the break detection. If, however, the shaft is connected to the dynamometer through an optional transmission, the dynamometer signal value would have to be adjusted accordingly. By defining the left and right speed signals as *required*, the break detection can be defined independent of the context where it will be used eventually.

Analogously to the hardware view, we also distinguish between interface and implementation in the software view: where in the hardware view the interface is defined by electrical plugs, flanges, or fittings, the interface in the software view is defined by the required input and output signals. A rounded block in the implementation diagram such as shown in fig. 5 a) and the port with the same name in the interface as shown in fig. 5 b) represent the same signal.

A function can be composed from other more elementary functions. These elementary functions eventually correspond to software executed by the automation system in real-time, i.e. in order to be able to transform a model, the targeted automation system must provide a matching implementation for each of the required functions. For example, in our case the automation system must know how to execute a delta function, and a reaction function. The automation system, however, does not need to have an implementation for composite functions such as the shaft break functionality.

Not all functions are consumers of signals such as the break detection. The engine, and the dynamometer components, for example, both come with a sensor to measure their speed. Fig. 6 shows part of the software functionality associated with the engine. It produces a crank speed output signal, as shown in the interface definition b). The implementation model a) shows the origin of the signal. It is produced by a specific kind of element that represents the physical phenomenon, acquired by a sensor and an I/O device. The engine's software functionality makes no assumptions about *how* this physical phenomenon is acquired, besides that it comes through some I/O functionality. The example also shows an additional reaction function, to express that the automation system has to monitor the speed of the engine and shut down the system in case the engine's maximum speed is exceeded.

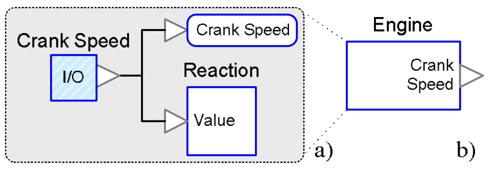


Fig. 6. Engine function's software view implementation a) and interface b). The implementation introduces the crank speed signal through a specific kind of model element representing I/O.

A signal defined by a function's implementation model can thus be made available to the outside through its interface, and at the same time the signal can be used internally for functionality associated with the engine function. The functionality associated with the dynamometer is analogous and thus is skipped here

C. *Integrating Hardware and Software Views*

The key to our modeling approach is not the graphical syntax used for the hard- and software aspects. It is also not the rich semantics of the language, although notions for sensors, plugs, and flanges, or software functions certainly are important for providing an intuitive modeling environment. More important is how we associate required standard software functionality for hardware components.

Continuing with the shaft example, fig. 7 picks up the interface definition of the shaft from fig. 3 and the break detection from fig. 5: a) and b) both are different *views* on the *same* model element, i.e. there is only *one* model element called shaft in the library. We thus use the term *component* for both, the *hardware* component, as well as its associated *required software* functionality.

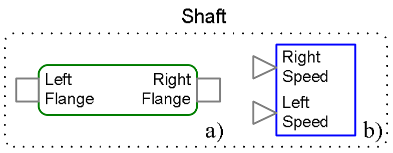


Fig. 7. Shaft hardware a) and software b) view. Both views together define the interface of a shaft component.

The hardware aspect of a shaft interface comprises the flanges, while the software aspect comprises the required speed signals and a model describing how the break detection is done in terms of more elementary software functions provided by the automation system software.

Obviously, a shaft is a simple mechanical component that can certainly not execute any software, yet we define software functionality with its model. Note that the software implementation model does not say where the functions are actually executed. In an automation system with multiple execution nodes, the delta function may be executed on another node than the reaction function. It is the model transformation system's task to find a suitable distribution.

Other components that are part of a test system such as the engine and the dynamometer are defined analogously to the shaft. The definition of a component such as the shaft containing both, hard- and required software functionality, is where the main advantage of our approach lies: since the shaft comes with required software functionality predefined, a user cannot forget it.

Note that the naming "Left Flange" / "Left Speed" and "Right Flange" / "Right Speed" suggests a semantic relation between the ports in the hardware and software views, but in general such a direct relation does not always exist.

Not all model elements need hard- and software aspects, the delta and reaction functions, for example, both represent pure software functionality. Considering the software functionality of the test system itself, the model from fig. 4

has to be extended accordingly. Fig. 8 shows part of the test system's software model containing the engine, the shaft, and the dynamometer components: these components may not only be used in the hardware model, but, since components also have a software view, they can be used in the test system's software model. This means that in general the software functionality can be implemented in terms of elementary or composite functions, as well as software interfaces of sub-components.

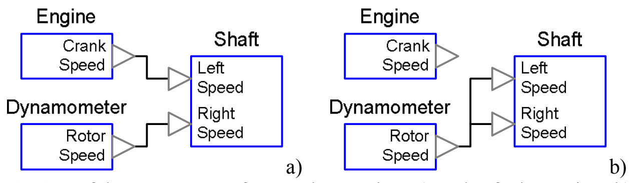


Fig. 8. A useful test system software view variant a) and a faulty variant b). The user of the predefined model elements decides how to connect them.

The example shows that two variants that are possible: just as users decide to connect the engine's flange to the shaft's left crank flange, and the shaft's right flange to the dynamometer's rotor flange, they decide how to connect the components in the software model. They, however, only have to make connections between the interfaces of the components, for example from the engine's crank speed output signal to the shaft's left speed input signal.

Users usually do not have to care about the implementation models of engine, dynamometer and shaft anymore, since they were already defined beforehand by the respective domain experts. Users describing a test system could, for example, not forget the break detection or how to implement it correctly, since it is automatically included in the model and the modeling environment checks that its required signals are connected. All they have to do is to provide all input signals with compatible output signals. As fig. 8 b) suggests, users may still erroneously choose to connect the dynamometer's rotor speed to both of the shaft's input signals. This way, however, the break detection would not work anymore, since in our case, its implementation assumes to be provided with two different speed signals. The implementation model is thus also the ultimate documentation of the functionality that helps users in understanding the functionality of an element by *zooming in*.

D. Ensuring Mandatory Model Elements via Template/Usage

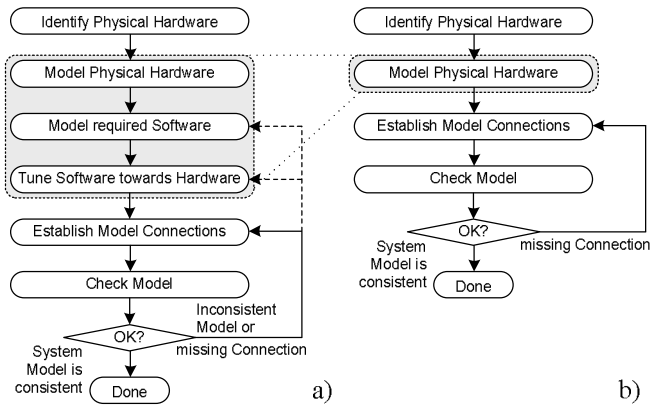
In section II we stated that our modeling environment must support defining parts of the model, such as the break detection mandatory, i.e. users must not be able to remove such functionality. We do this by virtue of our modeling language's implementation which has a special built-in relation between a component that serves as a *template* and its *usages*. This way our modeling environment provides flexibility similar to prototypical programming languages such as SELF [12] but with a more restricted semantics [13].

A component such as the shaft in fig. 7 defines certain structural features such as the flanges, or input signals. Since it is used as the template for the shaft in the test system model shown in fig. 4, and fig. 8, the modeling languages semantics prevents a user from arbitrarily changing the structural features in the usage; a shaft used within a test system model still has a left and a right flange. The semantics of templates and usages, however, allows for adding *new* ports, e.g. additional input or output signals. The semantics of this relation also includes the subcomponents: subcomponents defined by the template cannot be removed or changed at the usage, only *new* subcomponents can be added. For example, the shaft component usage in fig. 8 may include additional functions used for checking the maximum speed.

Besides structural features, the relation between templates and its usages has an effect on properties, too. For example, a shaft usually has a maximum rotational speed it can tolerate. The shaft template, however, might specify only that there *is* a maximum speed, and only at its usage a concrete value is provided.

E. The Configuration Process

The modeling language presented here supports integrated modeling of hardware- and software aspects. While this is a powerful feature on its own, the ultimate goal of our effort is to improve the process of configuring the automation system. To this end, the language offers a substantial potential by mitigating multiple sources of error. Fig. 9 illustrates how this is expected to influence the configuration process, in comparison to a conventional modeling approach with separate hard- and software models.



Identify Physical Hardware
Model Physical Hardware
Model required Software
Tune Software towards Hardware
Establish Model Connections
Check Model
OK?
Inconsistent Model or missing Connection
System Model is consistent
Done
a)
Identify Physical Hardware
Model Physical Hardware
Establish Model Connections
Check Model
OK?
missing Connection
System Model is consistent
Done
b)

Fig. 9. Automation system configuration process. Compared to the a) conventional process with independent hard- and software models, our b) improved process with an integrated hard-/software model mitigates multiple sources or errors.

Let us assume that we start with the physical test automation system already built. In the conventional approach, the configuration process starts with modeling hardware components of the system, such as the engine and the shaft. Each of these hardware components typically requires that the automation system provides some functionality so that the component can be operated safely. As a consequence, we have to create a software model that contains the corresponding functions. These functions, such the shaft break detection, now have to be adjusted to match the hardware characteristics. In our case, the break detection's threshold depends on the type of shaft used. The person preparing the model needs to know all these interdependencies between hardware and software, otherwise errors are inevitable.

This is where our approach shows its strengths: since for hardware elements the required software functionality is already defined and adjusted accordingly, the error-prone modeling steps can be skipped. In some situations one still has to adjust the software model, but on most occasions the predefined models are sufficient.

Once all elements representing hardware and software are added to the model, the modeling environment can validate the model and assist in detecting errors, for example by reporting missing connections. Here our approach also aids the user, since many connections are already predefined, and so it is less likely that a connection is missing.

IV. Model Transformation

The ultimate goal of our modeling approach is generating configuration data for existing automation system software. In our concrete case, this software uses configuration files. Entries in the configuration files directly represent a function executed by the software. For example, for each reaction entry, the software instantiates a corresponding function during startup of the system. This corresponding reaction function is then continuously executed in real-time.

The question now is how we can make use of the test system model to derive this configuration data. Actually, we promised that our modeling approach would free the users from dealing with this kind of low-level legacy aspects. The transformation thus must work automatically for each test system model that the users create, i.e. the transformation rules are defined only once and can be applied to different models. Remember that our modeling environment allows creating arbitrary models. It is thus unlikely that we could define a generic, global transformation rule that fits all possible models. Instead, transformation is split into smaller rules, each associated with a component. As shown in fig. 10 in addition to the hard- and software views, a component such as the shaft also includes a third aspect: the transformation rules for supported automation systems. The transformation rules are however not accessible to the end users.

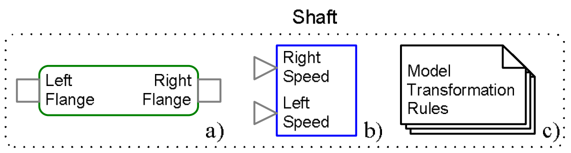


Fig. 10. The shaft complete shaft component.. A component can in general comprise a) hardware, b) required software, and c) transformation rules describing how the actual configuration data is created.

V. Implementation Status and Future Work

The approach described in this paper is already implemented in a modeling environment and at a state where it can be used by end users. Developers at our industry partner's office are now making the environment production ready. Meanwhile, domain experts of our industry partner use this environment to define a library of components necessary for describing the first concrete engine test automation system of one of their customers. Hand-in-hand with the components, transformation rules are defined. The library contains trivial model elements such as predefined electrical plugs, as well as complex model elements describing measurement devices. Experience so far shows that it is not always obvious which functionality should be associated with certain hardware component and that the design often requires multiple iterations. Certain aspects of the modeling environment, however, are still not finished yet, especially regarding I/O systems or model validation.

The relation between a template and its usages described in section III.D ensures that e.g. software functionality associated with a hardware component can not be removed. In terms of product line engineering, the relation thus allows describing mandatory features of a component. Sometimes, however, this behavior is too strict: instead, a component such as the shaft may come with multiple variants of possible break detection functions, from which the user may choose one, multiple, or none. We are thus planning to incorporate support for variability into our modeling environment [14] that respects the language's semantics.

References

[1] A. J. Martyr, and M. A. Plint, *Engine Testing: Theory and Practice*, Butterworth Heinemann, June 2007

[2] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*, Morgan Kaufmann, February 2008

[3] The MathWorks™, MATLAB®/Simulink®, http://www.mathworks.com/products/simulink/

[4] EPLAN Software & Service GmbH & Co. KG, EPLAN, http://www.eplan.com/

[5] C. Reichmann, M. Kühl, P. Graf, and K. D. Müller-Glaser, "GeneralStore - A CASE-Tool Integration Platform Enabling Model Level Coupling of Heterogeneous Designs for Embedded Electronic Systems", *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*, pp. 225-232, IEEE Computer Society, May 2004

[6] OMG Unified Modeling Language™ (OMG UML), Superstructure, OMG, February 2009

[7] The Föderal Initiative, http://www.foederal.org/

[8] Mind8 GmbH & Co. KG, Mind8, http://www.mind8.com/

[9] D. M. Weiss, and C.-T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley Professional, August 1999

[10] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley, July 2004

[11] D. Dhungana, P. Grünbacher, and R Rabiser, "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling", *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems*, pp. 119-127, Kindai Kagaku Sha, January 2007

[12] D. Ungar, and R. B. Smith, "Self: The power of simplicity", *SIGPLAN Notices*, vol. 22(12), pp. 227-242, ACM, December 1987

[13] T. Aschauer, G. Dauenhauer, and W. Pree, "Multi-Level Modeling for Industrial Automation Systems", *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 490-496, IEEE Computer Society, August 2009

[14] G. Dauenhauer, T. Aschauer, W. Pree, "Variability in Automation System Models", *Formal Foundations of Reuse and Domain Engineering*, pp. 115-125, Springer Verlag, September 2009