Applying Software Product Line Techniques in Model-based Embedded Systems Engineering

Andreas Polzer and Stefan Kowalewski
RWTH Aachen University, Aachen, Germany
{polzer | kowalewski}@embedded.rwth-aachen.de

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
University of Limerick, Limerick, Ireland
goetz.botterweck@lero.ie

Abstract

This paper addresses variability in the domain of software-based control systems. When designing product lines of such systems, varying sensors and actuators have to be used and parameterized, which in turn requires adaptations in the behavior of the microcontroller. For efficient engineering these adaptations should be performed in an systematic and straightforward manner.

We tackle these challenges by using a Rapid Control Prototyping (RCP) system in combination with model-based development techniques. In particular, we modularize the parametrization of components into a separate configuration, which is isolated from the model that defines the controller behavior. Hence, during adaptations the model can often remain unchanged, which significantly reduces the turnaround time during design iterations. The approach is illustrated and evaluated with a parking assistant application, which is tested on our experimental vehicle, where it performs automatic parking maneuvers.

1. Introduction

Microcontroller-based control systems interact with their environment via sensors and actuators. When dealing with *families* of such systems, which are targeted for specific market segments or contexts, different types of actuators and sensors with varying parameters are used.

However, when modifying or replacing such components, one has to take into account that this requires adaptations in the behavior of the controller. Consequently, approaches that allow variations in both (1) the selection and parametrization of actuators and sensors and (2) the corresponding behavior of the controller are required. Moreover, these adaptations should be performed in an systematic and straightforward manner.

In this paper, we address these challenges with an engineering approach that uses a Rapid-Control-Prototyping System in combination with techniques from Software Product Lines (SPL) and model-based development. In particular, we isolate variable parametrization of components into a configuration file separated from the Simulink model which describes the controller behavior.

To allow for derivation of products – with both Simulink model and configuration file synchronized – the available product configurations are described using a feature model and linked to the implementation with a product line tool. Hence, when a particular product is configured, both the Simulink model describing the behavior of the controller and the configuration file describing the corresponding actuator/sensor configurations are adapted consistently.

We choose Simulink as a modeling and simulation environment since it is effectively an industry standard. Consequently, well-known model-based techniques can be applied, e.g. when testing the application in a simulation before moving on to the real execution environment.

As a consequence of the encapsulated configuration, the model can remain unchanged in many cases and the turnaround time between modifications and the availability of the executable/testable application is reduced. Additionally, the transfer from simulation to testing using a prototyping hardware is straightforward since no changes are necessary within the model.

The approach is evaluated using our rapid prototyping test-bed for automotive embedded systems. For this, we use a sample product line for parking assistance applications, which is fully implemented and can be tested on our scaled model cars.

MOMPES'09, May 16, 2009, Vancouver, Canada
978-1-4244-3721-4/09/$25.00 © 2009 IEEE

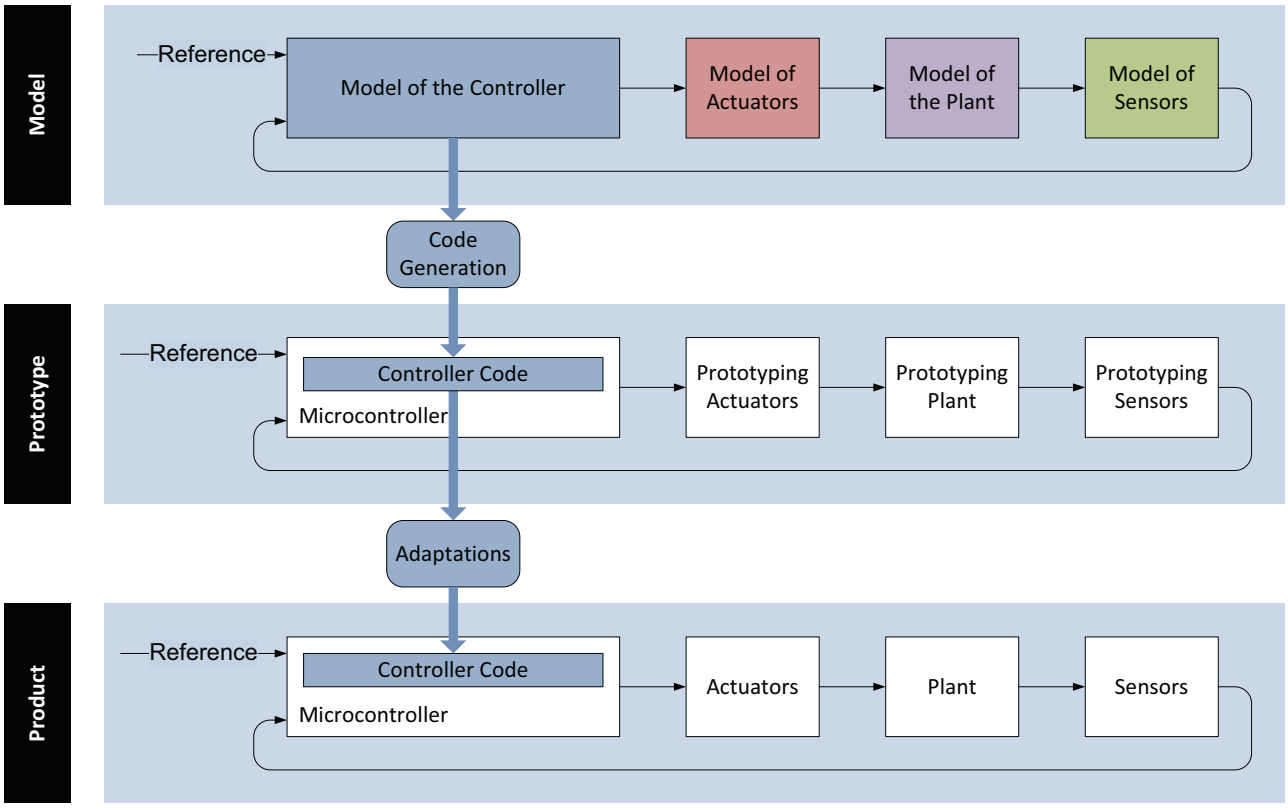

Figure 1. Model-based development of control systems.

2. Research problem

2.1. Software-based control systems

Software-based control systems are used in very different scenarios, for instance robots, toasters, airplanes, and machine tools. A prominent usage area for such systems are automotive applications, such as engine management, safety functions, or driver assistant systems [1].

When developing software-based control systems, it is common engineering practice to use *Rapid-Control-Prototyping (RCP)* systems. The overall development process can be structured into three phases (see figure 1).

First, the engineer designs envisioned controller by describing a *Model* of the envisioned system. Typically he uses modeling environments like Matlab/Simulink. To allow for simulation of the whole system, models of other system components (such as actuators, the plant, and sensors) are included as well. By experimentation and simulation of test scenarios the behavior and the quality of the implementation (as described by the model) can be evaluated and improved. During these improvements the engineers typically aim to optimize the system with respect to certain key values, e.g., reaction and settling time.

Once a sufficient quality level has been reached, the model will be transformed into an implementation which can be tested in a *Prototype* environment. Usually this is done via automatic code generation and compilation. When transferring the implementation into the prototype environment, some components which where simulated in the preceding phase (e.g., actuators, the plant, sensors), are replaced by real components. In other words, only the code for the controller itself is generated and deployed to the prototyping environment. Subsequently, the implementation of the new system can be tested and evaluated, now in a test environment that is even more realistic than the simulation. This helps to test the implementation with the real interfaces and the corresponding timing issues.

Finally, the implementation is checked off and delivered as a product. Depending on the concrete case, this might be the deployment into one assigned production environment or the production of large unit numbers.

To understand the motivation behind our research, it should be noted that in a typical industry project the engineers iterate over the first two phases (modeling and testing in an prototyping environment) many times, until the product design is finalized. Moreover, if the control system has to be adapted to a new environment, e.g., with a modified

plant behavior and different sensors/actuators and controller parameters, this procedure has to be repeated to allow for adaptations.

Consequently, we are looking for an approach that (1) improves this engineering process of modeling and prototype testing and (2) eases adaptation of a control systems, especially with respect to variability in sensors/actuators and control parameters.

2.2. Concrete case: parking assistant

To illustrate the research problem we will now introduce a concrete sample case, a parking assistant. The schematic architecture of the application is shown in figure 2.

The overall application is controlled by a microcontroller, which receives input from a variety of sensors. For instance, it gets information on vehicle speed, distance to obstacles, and the vehicle direction. In addition it receives commands given by the drivers. For the prototype environment these commands are set via a wireless radio control (RC) and received by the car via an RC receiver. At the same time the controller has to act on its environment by several actuators including an engine throttle, brakes, and the steering.

The sensors have different characteristics which have to be taken into account when determining the requirements for the controller. For instance, one measurement of an ultra sonic sensors takes about 60 milliseconds. Thus, if we want to measure distances during each computing step, a cycling time longer than 60 milliseconds is necessary. Similarly, this cycling time influences the speed of reaction to user commands.

Because of these dependencies, whenever usage scenario is modified, it is necessary to adapt the sensors and actuators. In many cases, the main functionality of the controller can remain unchanged. For instance, in our parking assistant – regardless of adaptations of sensors/actuators and their parameters – the "strategy" for parking and the corresponding algorithm remain unchanged. The procedure is executed in three phases:

First, a parking spot is identified and measured. During this phase, the car is manually driven by the user. Simultaneously the side distance sensors and velocity sensors are searching for an adequate parking spot. In the default mode, the assistant is activated as soon as a parking spot is detected on the passenger (i.e., right) side of the vehicle. During each computing cycle the reclined distance is computed by integrating the velocity. A parking bay is deemed sufficient, if both length and depth fulfill the constraints, which were set earlier during parametrization.

In the second phase, after a parking spot is detected, the controller waits 10 seconds for a confirmation by the driver. Given this confirmation, the parking assistant takes over the control of the actuators. At any time the user can interrupt the automatic parking and take back control by using the break or steering.

Once the parking starts, the car is driven (by the controller) into an initial position. Then the parking assistant goes backwards into the parking bay. To this end, a trajectory is computed and followed during parking. Finally, the car is aligned to the side of the road and other obstacles by using distance sensors and optionally (if available) the direction sensor.

3. Our approach

In the preceding sections we defined the research problem that we focus on (variability in embedded systems) and a concrete example case (a product line of parking assistance applications). We will now present our approach which we developed to make the engineering of such applications more efficient.

3.1. Hardware abstraction

As a basis for the engineering of such applications, we developed an architecture for a *Rapid-Control-Prototyping (RCP)* system called *VeRa* (Vemac Rapid-Control-Prototyping System). To simplify the handling of varying product configurations we introduce an *Hardware Abstraction Layer (HAL)* which (1) isolates the core application from sensors and actuators and (2) manages data sent to actuators or coming from sensors.

The encapsulation of hardware-specific functionality introduced by the HAL has several advantages: (1) the core application is shielded from hardware-specific details and changes in implementation details. (2) It allows us to introduce a variability mechanism for sensors and actuators, which reads product-specific parameters from an XML-based configuration file. (3) The transfer from simulation to testing (using a prototype) is straightforward since no changes are necessary within the model.

Integrating varying sensors and actuators is a major task when adapting a given controller model for a new system. For instance, changes in scales of value domains or modifications in the surrounding environment have to be reflected in the behavior of the controller. Hence, it is desirable to provide an simple way to adapt sensors and actuators via the abstraction layer.

The abstraction layer relies on the fact that information provided by the sensors can be described in general terms using physical units. Similar applies to information sent from the controller to actuators. With this approach it is possible to use a common interface for varying applications. During model-based development, this common interface is

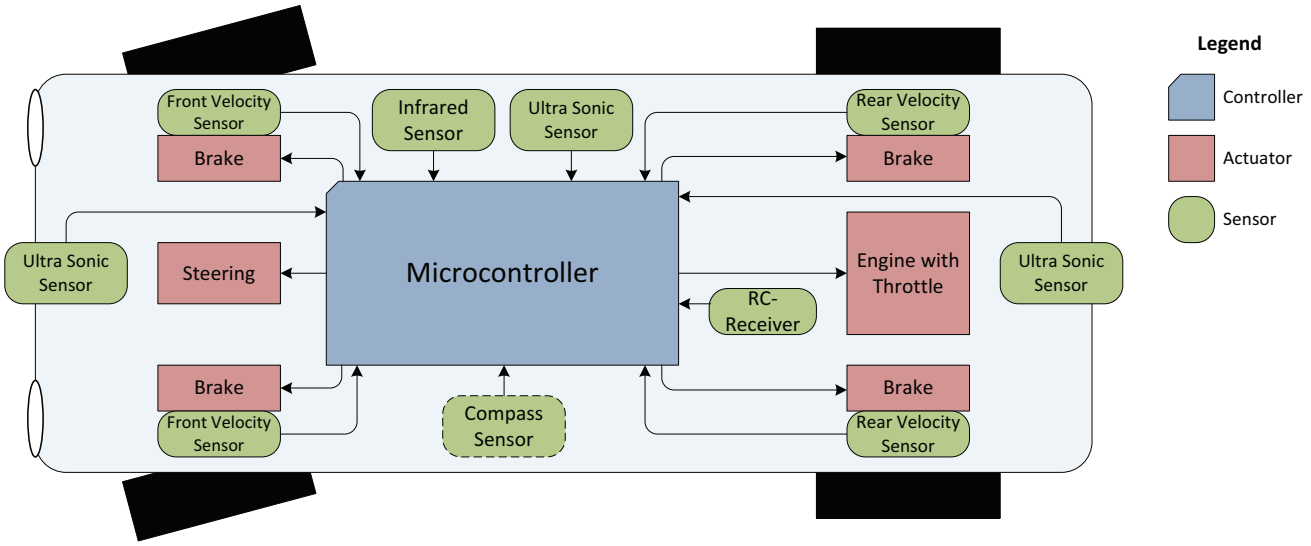

Figure 2. Components and connection of the experimental vehicle.

used to connect the controller with other components. Consequently, we can adapt the used sensors and actuator to a new scenario without modifying the application model. The configuration file provides all required information on the product-specific settings for sensors and actuators.

We will now explain the sequence of activities and data flow in our approach in more detail (see figure 3). Similar to common software product line frameworks [5, 12], the approach is horizontally structured in two layers, Domain Engineering (developing the product line) and Application Engineering (deriving the product). Vertically we distinguish three areas: (1) modeling of features, (2) mapping features to implementation components, and (3) modeling the implementation itself. For the implementation we also differentiate handling within pure::variants and description by textual DSLs, thus we end up with four vertical areas in total.

We will now look at the activities of Domain Engineering and Application Engineering in more detail.

3.2. Domain Engineering

Domain engineering starts with the process of *Feature Analysis* ❶ which produces a *Feature Model*, similar to the one discussed earlier and shown in section 4 using the tool pure::variants. The identified features are implemented ❷ using common practices from embedded systems engineering. For instance, in the case of model-based development, features are implemented in Matlab/Simulink. In our particular approach, we extended this with custom components (i.e., Simulink block types) representing sensors and actuators.

Later during Application Engineering, we will use *negative variability* [14] to derive products, i.e., product-specific artefacts will be created by copying product line artefacts and selectively filtering out certain components, based on a given configuration. To prepare for this Application Engineering procedure, earlier in Domain Engineering we have to map features to the *corresponding* implementation components.

To this end, feature implementations (given as Simulink models and XML configuration files) are imported using the plug-ins pure::variants Simulink Connector and pure::variants XML cond ❸. This import process creates *Family Models* 🅱, which represent the implementation. Mappings between features and implementation elements are then described by conditional expressions, which are stored in the *Family Models* 🅱. For instance the expression `hasfeature(RearBrake)` will evaluate *true* whenever the feature `RearBrake` was selected and the corresponding components, which were marked with this expression, will be included in the particular product implementation.

It should be noted that different features are mapped to different *Family Models*. Features, which influence the XML configuration, are mapped to the *XML Family Model* whereas other features correspond to implementation components in the *Simulink Family Model*.

Another artefact created during Domain Engineering is the *Transformation Configuration* 🅲, which later controls the generation process performed in Application Engineering.

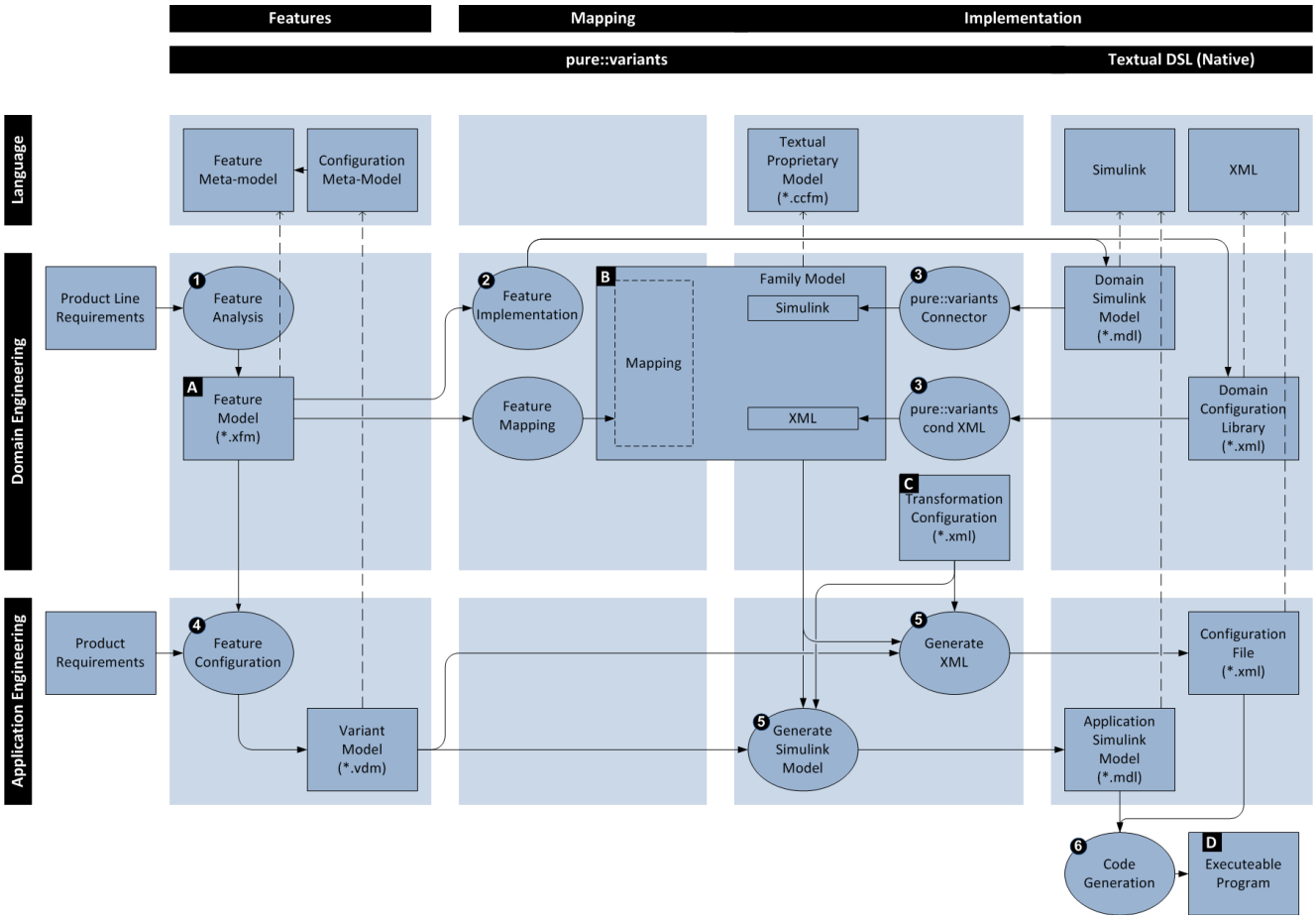

Figure 3. Overview of our approach.

3.3. Application Engineering

In the end, the goal of Application Engineering is to create a new product which fulfils the requirements of a particular customer. In the case of model-based controllers, reasons for developing a new product could be the adaption of the controller for a new system or the use of different hardware caused by new requirements, e.g., replacing an infrared sensor with an ultrasonic one.

The procedure for deriving a product is as follows (see the lower layer in figure 3): Given the *Product Requirements* the engineer performs *Feature Configuration* ❹ to identify corresponding capabilities of the product line (i.e., features), which can be used to cover these requirements. The configuration is stored as a pure::variants *Variant Model*.

The *Variant Model* does not store the complete configuration information. Instead it contains only the made decisions; other information is given by references to the domain level *Feature Model* 🅐, *Family Model* 🅑 and *Transformation Configuration* 🅒. Consequently, changes in those models implicitly affect the variant model as well.

The set of features, which was selected in the configuration process earlier ❹ and stored in the *Variant Model* is used as input for the transformations, which create the product: The first step of this process is the verification of restrictions and relations of the configuration. Then, the transformation operations specified in the *Transformation Configuration* 🅒 are executed. In our sample case the transformation process includes two types of related artefacts. On the one hand, it derives the Application-specific Simulink Model On the other hand, it derives the XML configuration file, which parameterizes the library of sensors and actuators.

As we are applying negative variability here, this derivation creates new artefacts (Simulink model, configuration file) by copying the corresponding domain-level artefacts and removing all elements (e.g., Simulink blocks) which are not required to implement the set of features given by the product configuration.

Finally, as a last step of product derivation *Code Generation* ❻ is performed using the Modeling and Simulation tools Matlab/Simulink and Realtime Workshop. During this process the model and configuration generated earlier are processed to generate source C code. As a result we get an executable program, which can be deployed onto the RCP Hardware.

4. Feature model

To capture the available configurations options and the constraints on them, we use a feature model. See figure 4.

The `Parking Assistant` application consists of `Actuators`, `Sensors`, a `Controller`, the `Plant`, `User Commands` and helper functionality for `Debugging`.

Most of the `Actuators` are mandatory features (`Throttle`, `Steering`, `Front Break`). Merely, the `Rear Brake` is optional.

For `Sensors` there is more variability. For some `Distance` sensors, we can choose whether to use `Infrared` or `Ultrasonic` technology or if they are present at all. For the `Velocity` sensor, we can choose between a standard `Impulse` sensors, which are mounted at the front wheels, and an `Hall Impulse` sensors at the rear wheels. Moreover, depending on the particular product we can decide if we need a `Direction` sensor.

The `Controller` includes the core algorithm for the parking assistant. Here, we have to decide whether we require an algorithm that takes direction into account.

The `Plant` is optional, since we might want to simulated the whole application within the modeling tool, whereas in other situations we might want to exclude it. Compare the different levels (Model, Prototyping, Deployed Product) in figure 1.

The `Debug` feature allows to toggle scopes and other debug mechanisms in the model. For instance, this provides tracking information during both simulation and execution on the prototyping test-bed. In the latter case, the data can be polled from a diagnosis PC via a serial connection.

The `User Commands` provide functionality to gather commands from the user. Currently, in the prototype environment this is done via remote control. In case of simulation the commands are given by a gamepad connected to computer.

5. Evaluation

In the preceding sections, we presented an approach for product lines of microcontroller-based control systems, which combines the common model-based design with modularized configurations.

5.1. Experiments with a model car

To evaluate our techniques in a realistic scenario we implemented the parking assistant (described earlier in section 2 and deployed it onto our *Automotive Experimental Vehicle (AEV)*. A photograph of the model car can be seen in figure 5). It is equipped with our Rapid-Control-Prototyping (RCP) system called *VeRa*, which software architecture was developed in our group in collaborations with industry and research partners.

The logical architecture of VeRa, including the data flow between components, corresponds to the schema shown in figure 2. The VeRa is connected with sensors to measure distances in the front, in the rear, and on the right side. It would be easy to add distance sensors on the left side, but this was not necessary for our evaluation. We are experimenting with varying infrared and ultrasonic distance sensors and sensor parameters. To measure the velocity both front wheels are equipped with forked light barriers, the rear wheels with Hall sensors. Driver control is simulated via commands given via a remote control.

These sensors are connected to the RCP system through various interfaces: The ultra sonic sensors are connected via a digital I^2C bus. The infrared sensors provided their measurement via analogous signal (i.e., voltage). Velocity is measured by counting electrical impulse within a certain time period. User commands incoming from the RC receiver are provided to the controller as PWM (Plus Width Modulation) signals. Because of these varying signal types, different hardware components, drivers and data formats are required to get the information from the sensors.

Fortunately, using the abstraction layer (discussed earlier in section 3.1), we can shield the ore application from hardware dependencies. Hence, the core application could be designed by modeling and simulation in Matlab/Simulink, while abstracting from hardware details. For instance, we do not have to care whether the distance information is provided by an ultrasonic or an infrared sensor. Finally, the code is generated with Realtime Workshop and deployed.

5.2. Discussion

Although first results with the approach are promising, it also has some limitations. The techniques used to bring sensors and actuators into the hardware abstraction layer require additional overhead for integration and development of glue code. Consequently, this approach is not reasonable for “quick hacks” where new component types have to be used as fast as possible. It should be noted that this limitation applies to *new types* of sensors/actuators. As long as the engineer sticks to types of sensors/actuators that have been used before (i.e., where an integration into the HAL was already done) our approach will actually speed up his

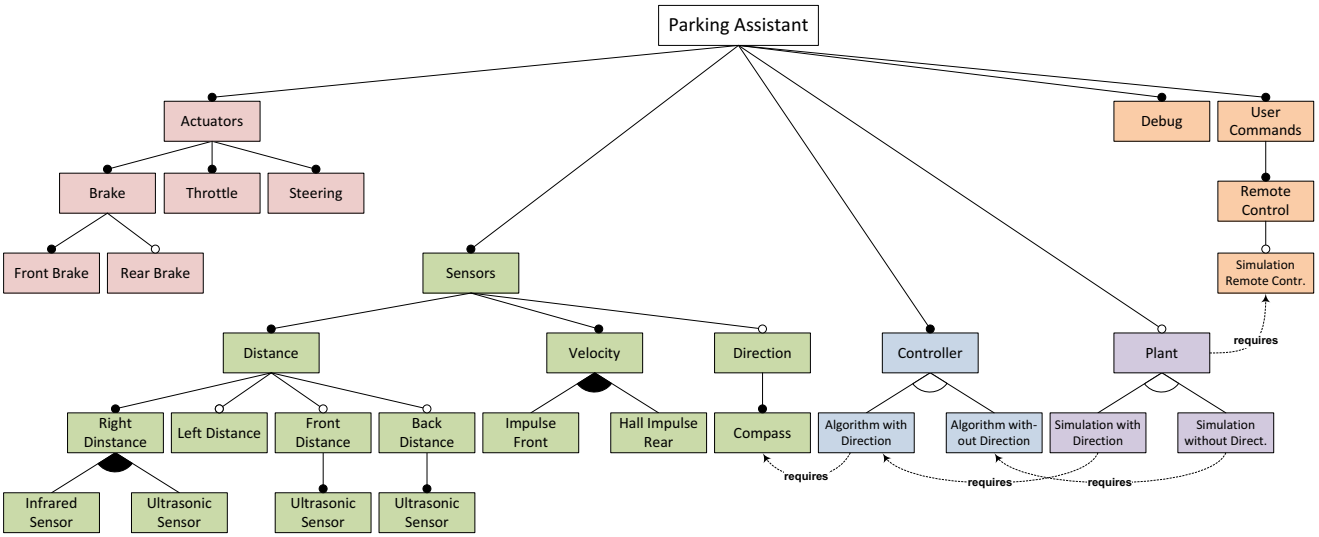

Figure 4. Feature model of the parking assistant product line.

development efforts.

During development the engineer has to move back and forth between modeling/simulation and the prototype environment (the upper two levels in figure 1). In our approach, the whole system including controller, actuators, sensors, and plant can be simulated. In particular, the simulated sensors imitate quantization errors and timing constraints to allow for a realistic simulation of their real electronic counter parts. When switching to the prototype environment the simulated sensors/actuators are automatically replaced by drivers which connect the controller to the real components. Overall, this provides an seamless transition from simulation to the prototyping environment.

Since we modeled configuration choices by a feature model (figure 4) and mapped the features to the corresponding implementation components we are able to configure and generated a product relatively straightforward. One of the reasons, why this is possible at all, is the hardware abstraction layer which reduces dependencies between chosen options (e.g., sensors) and the core application.

Another challenge, which we have to address is the consistency between the varying models. For instance, the parking assistant product line has an *optional* direction sensor. If this sensor is present in the particular product, it can be used to monitor the alignment of the car in the parking bay with higher accuracy. Otherwise, without this data, the parking assistant has to try a "best effort" approach to get the car aligned to the parking bay. The implementation of these different strategies causes variability in both the behavior of the controller (i.e., the Simulink model) and the sensor configuration (i.e., the configuration file). For varying configurations, we want to ensure consistency between the involved artefacts. In the presented approach this is supported by modeling variability options in exactly one artefact (the feature model) which influences all others.

At the current state of research we do no yet have quantitative data on the improvement of engineering practice (e.g., "20 percent faster"). However, the experiments with the parking assistant on the VeRa platform indicate that in general our effort of integrating rapid prototyping of embedded controller applications and product line engineering was successful, in the sense that (1) we have a smoother transition between modeling and testing in the prototype environment, resulting in shorter turnaround times during engineering iterations and (2) variability is modeled in just one model, which in turn influences all other artefacts in a consistent way. Together, this allows for more efficient development of software-based controller applications with varying sensor and actuators configurations.

6. Related work

The research presented is based on earlier work [4], which we extend here by (1) addressing variability in microcontroller-based control systems with a combination of Simulink models and configuration files, (2) presenting concepts for the isolation of variability cased by the configuration and by (3) evaluating the approach to a more complex product line with prototypes based on the Vemac Rapid-Control-Prototyping System.

When dealing with variability in domain-specific languages a typical challenge is the mapping of features to their implementations. Here, Czarnecki and Antkiewicz [6] used a template-based approach where visibility conditions

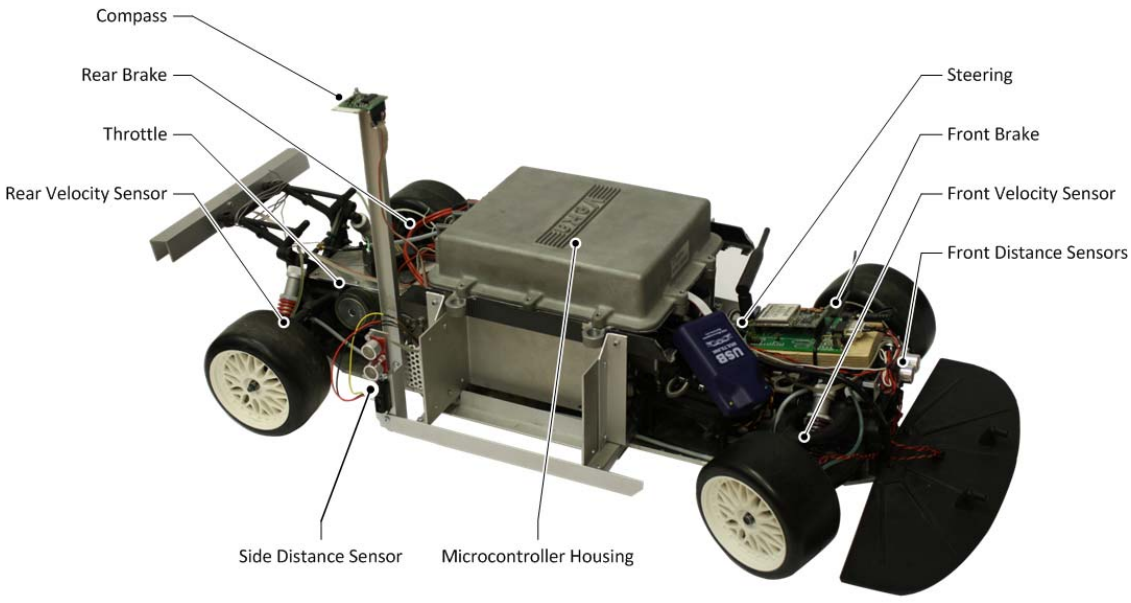

Figure 5. The model car of the VeRa Rapid Control Prototyping platform.

for model elements are described in OCL. In earlier work [2, 3], we used mapping models and model transformations in ATL [7] to implement similar mappings. Heidenreich et al. [10] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [8].

The approach discussed in this paper is partly based on mechanisms provided by pure::variants, in particular the Simulink connector [13].

Voelter and Groher [14] used aspect-oriented and model-driven techniques to implement product lines. Their approach is based on variability mechanisms in openArchitectureWare [11] (e.g., XVar and XWeave) and demonstrated with a sample SPL of home automation applications.

7. Conclusions

In this paper we focused on the problem of introducing Product Line Engineering into the domain of software-based controllers.

To address this challenge we introduced a Hardware Abstraction Layer (HAL), which allows to exchange and adapt hardware components without changing the model-based implementation. To support consistency we describe the product line with *one* model (the feature model) which affects all other artefacts (Simulink model and HAL) in a consistent way. We evaluated our approach by implementing a parking assistant with varying product-specific configurations and sensors.

In our current work we are currently preparing to use more capabilities of model-driven frameworks such as EMF [9], GMF [9], and openArchitectureWare [11]. This includes the use of Ecore-based meta-models for the involved domain-specific languages (e.g., Simulink) and mechanisms for processing models (e.g., model transformations and model weaving). Consequently, this will allow us to improve the handling of large and complex product lines, for instance by providing better approaches for feature-implementation mappings as well as concepts for complexity handling by abstraction.

References

[1] D. Abel and A. Bollig. *Rapid Control Prototyping*. Springer, Berlin, Heidelberg, 2006.

[2] G. Botterweck, K. Lee, and S. Thiel. Automating product derivation in software product line engineering. In *Proceedings of Software Engineering 2009 (SE09)*, Kaiserslautern, Germany, March 2009.

[3] G. Botterweck, L. O'Brien, and S. Thiel. Model-driven derivation of product architectures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007)*, pages 469–472, Atlanta, GA, USA, 2007.

[4] G. Botterweck, A. Polzer, J. Palczynski, R. Mitsching, and S. Kowalewski. Applying DSL technologies to realise vari-

ability in embedded systems software (submitted for review). In *IFIP Working Conference on Domain Specific Languages 2009*, Oxford, England, UK, July 2009.

[5] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA, 2002.

[6] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, 2005.

[7] Eclipse-Foundation. Atl (ATLAS Transformation Language). `http://www.eclipse.org/m2m/atl/`.

[8] Eclipse-Foundation. EMF - Eclipse Modelling Framework. `http://www.eclipse.org/modeling/emf/`.

[9] Eclipse-Foundation. GMF - Graphical Modelling Framework. `http://www.eclipse.org/modeling/gmf/`.

[10] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: mapping features to models. In *ICSE Companion '08: Companion of the 13th international conference on Software engineering*, pages 943–944, New York, NY, USA, 2008. ACM.

[11] openarchitectureware.org - official open architecture ware homepage. `http://www.openarchitectureware.org/`.

[12] K. Pohl, G. Boeckle, and F. van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, New York, NY, 1st edition, 2005.

[13] Pure::systems. pure::variants Connector for Simulink. `http://www.mathworks.com/products/connections/product_main.html?prod_id=732`.

[14] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC 2007*, Kyoto, Japan, September 2007.