Interactive Configuration of Embedded Systems Product Lines

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

Andreas Polzer and Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
Email: {polzer, kowalewski}@embedded.rwth-aachen.de

***Abstract*—This paper addresses product configuration and product derivation in product lines of embedded systems. We show how domain-specific languages (DSLs), which are used to describe the implementation of the product, can be translated into configurable models with formal semantics. This facilitates, tool support during configuration including (1) side-by-side visualization of features and corresponding implementation components, (2) automated reasoning to calculate consequences of the user's configuration decisions and (3) visual explanations for automatically calculated consequences. In addition, we discuss (4) how a completed configuration can be turned into a product-specific model in the domain-specific language, using negative variability and subsequent pruning of the implementation model.**

The approach is demonstrated for product lines of embedded systems using Simulink as an domain-specific language for the model-based engineering of embedded systems. We report on first evaluation results with a product line of parking assistant applications, including experimentation on a rapid prototyping platform with a 1:5 model car.

I. Introduction

Many approaches in Software Product Lines (SPL) structure the applied models into two areas (see figure 1): A model describing the available choices, e.g., a feature or variability model A_d and, one or more implementation models, which describe how these choices are implemented C_d. Usually these two types of models are mapped onto each other B to support further techniques.

During *Product Configuration* the user (i.e., a customer or an application engineer supporting him) decides which of the available product options to choose. In *Product Derivation*, he generates or assembles the product implementation that corresponds to these configuration decisions.

There are well-known techniques and tools for the interactive configuration of feature models, for instance in commercial tools such as *pure::variants* [1] or our earlier work on interactive configuration with formal semantics [2], [3].

Interestingly, during product configuration the developer typically configures the feature model A_d *only* – even though the mappings between the feature model and other SPL models are available B. Interaction with the implementation models C_d is usually *not* provided at this stage.

While there might be good reasons to abstract from the implementation deliberately (e.g., to hide complexity), the inclusion of other models in the interactive configuration process can provide additional benefits:

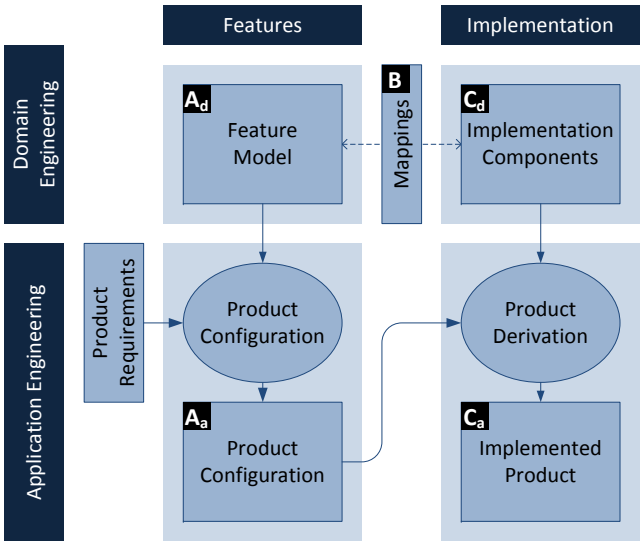

Fig. 1. Simplified Framework for Software Product Line Engineering.

1) The user can see (visual representations of) dependencies between features and the related elements in other models (e.g., an edge representing that the feature f_1 requires the component c_1).
2) When making configuration decisions (in the feature model) the user can immediately see consequences in related models (e.g., after the user selects the feature f_1, the tool automatically selects the component c_1).
3) Moreover, it is possible to apply configuration decisions in the implementation model first (e.g., indicating that the implementation component c_1 will not be available) and derive implications for the feature model from that (f_1 is no longer an option).

In this paper, we address this side-by-side configuration by integrating the implementation model into the configuration process. This allows us to provide the functionality sketched above. To further motivate our research, we will first use a sample case from product lines of embedded systems (section II). We will then explain our approach (sections III

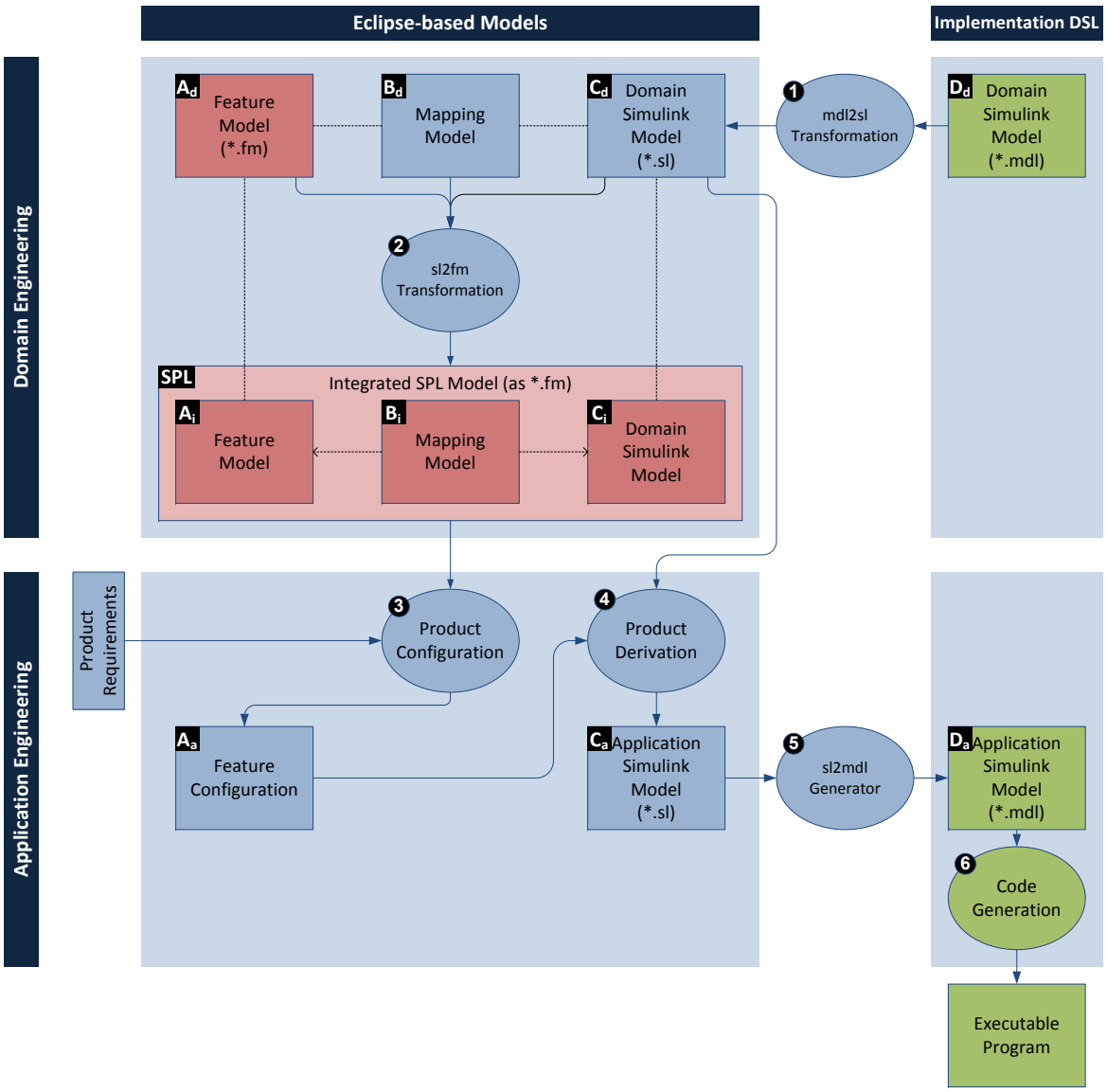

Fig. 2. Overview of our approach.

to VI). The paper concludes with a discussion (section VII), an overview of related work (section VIII) and some final remarks.

II. Sample Case: Embedded Systems in Control Engineering

As a sample case for this paper, we want to apply Software Product Line techniques to the domain of control engineering. Such control system are typically developed using model-based tools like Matlab/Simulink. They can be considered as (or implemented as) a special form of embedded systems.

The task of a *controller* is to influence an environment with *actuators* to achieve a certain behavior. To this end, the controller gathers information of the environment using *sensors*. With the information provided by the sensors the controller uses the actuators to influence the controlled environment. In many cases, the part of the environment that is the object of control (observed by the sensors, influenced by the actuators) is called the *plant*. The whole system (consisting of sensors, controller, actuators, and the plant) is called a *control loop* [4].

When developing the code for such a controller, the main requirements are the reaction time, the input-output stability and the control error. The behavior of the control loop depends on both the controller and the plant. To understand the behavior of the plant and the controller, models of both are developed in a first step. These models are improved using the results of simulation.

If the desired behavior is achieved, a next development step is performed. The model of the controller will be translated to source code and executed using a prototyping hardware. During this development step either a prototyping plant or a real-time model of the plant is influenced by the controller. In this development timing requirements can be observed and

tested.

During the third development step the controller code is executed on the real product hardware. This is the first time the real plant (and not the simulated one) is tested with the controller. Cost and safety issues are the main reasons for the late testing with the real plant. One important task during this stage is to optimize the controller. To this end, the controller has parameters that can adopted until the control loop finally meets the requirements.

To built such systems, model-based design is a common engineering practice. Simulink is a very well-known example of a domain-specific modeling language for embedded systems, including corresponding tools. Using such development frameworks is one way to tackle the increasing complexity.

While this is already a nice foundation, in an industrial context we require additional techniques that help us to build whole product lines of such systems. To this end, model-based design techniques for embedded systems are extended with mechanisms for variability and model-driven product derivation.

We discussed some concepts and techniques for this in [5], [6] where we extend domain specific implementation models with variability. In this paper we focus on how feature models and the related implementation models can be combined to support their integrated, interactive configuration.

III. Overview of our Approach

Before we explain the details of our approach, we first want to give an overview as an orientation for the reader, see figure 2. Similar to common SPL Engineering methods our approach can be structured into *Domain Engineering* (upper layer) and *Application Engineering* (lower layer).

The overall goal of this process is to turn a product line implementation D_d (upper right corner of figure 2) into an product-specific implementation D_a (lower right corner) and finally an executable program.

To support common techniques and processes from Embedded Systems Engineering, we integrated our approach with the domain specific language *Simulink*. Hence, the chain of processes ❶ to ❻ begins in the Simulink world (right-hand side of figure 2) moves over (❶) to the Eclipse-based Models (left-hand side), which are configured and processed. Finally, by code generation (❺) we return to the implementation DSL. In the following sections we will now discuss these processes in more detail.

IV. Domain Engineering

For the context of this paper, we will assume that most processes, which are necessary to create the product line artefacts (A_d to D_d) have already been performed. See [5] for more details.

Those processes that are of interest here, start off with the *mdl2sl transformation* ❶, which converts the implementation given in Simulink's native .mdl format D_d into an corresponding Eclipse-based implementation model C_d. Subsequently, we are able to map this model to the feature model and perform further processing with Eclipse-based frameworks, such as the numerous frameworks from the Eclipse Modeling Project (EMP) [7] or openArchitectureWare (oAW) [8].

To enable the integrated configuration of feature and implementation models, we transform ❷ these models and the mappings between them into an integrated SPL model **SPL**. The basic idea is to represent all configurable parts of the product line (feature selected? component present?) as one large feature tree, where different subtree represent different SPL models. So, for instance, we can have one subtree with the real feature model A_f and one subtree representing the configuration status of components C_f, as well as mappings and between them B_f. This enables us to interactively configure the whole product line within one integrated model.

This translation is realized by an model transformation in ATL (Atlas Transformation Language) [9]. It applies an semantic interpretation of the domain-specific concepts in the Simulink model, translating them into feature model elements, which make up the integrated SPL model **SPL**. Some rules for translation are shown in Table I for the Simulink model (translating from C_d to C_f) and Table II for the mapping model (translating from B_d to B_f).

Simulink C_d (concepts in the meta-model)	Representation within the integrated SPL model C_f
Simulink Model m	mandatory feature $f(m)$
System s	optional feature $f(s)$
Block b	optional feature $f(b)$
contained blocks/systems in blocks/systems	subfeatures
Line from block a to block b	$f(b)$ requires $f(a)$ but not vice-versa

TABLE I
Translation from Simulink to the integrated SPL model

Mapping B_d (concepts in the meta-model)	Representation within the integrated SPL model B_f
Feature f mapped to component c	$f(f)$ $requires$ $f(c)$

TABLE II
Translation from mapping model to the integrated SPL model

The elements created in the target model are fine grained elements of a feature model, which we call *feature model primitives*. Examples for such feature model primitives are f_1-*hasOptionalSubfeature*-$f2$, f_3-*requires*-f_4 or $f1$-*hasBeenSelected*. These primitives have clearly defined semantics, including the corresponding behavior of our `S2T2 Configurator` tool during interactive configuration. These semantics are given by further translation of the feature model primitives into propositional logic. For instance, f_3-*requires*-f_4 would be translated into $\neg f_3 \vee f_4$. Details of this translation can be found in [2].

In summary, the transformations provide the following result: We now have (1) an integrated model presenting features,

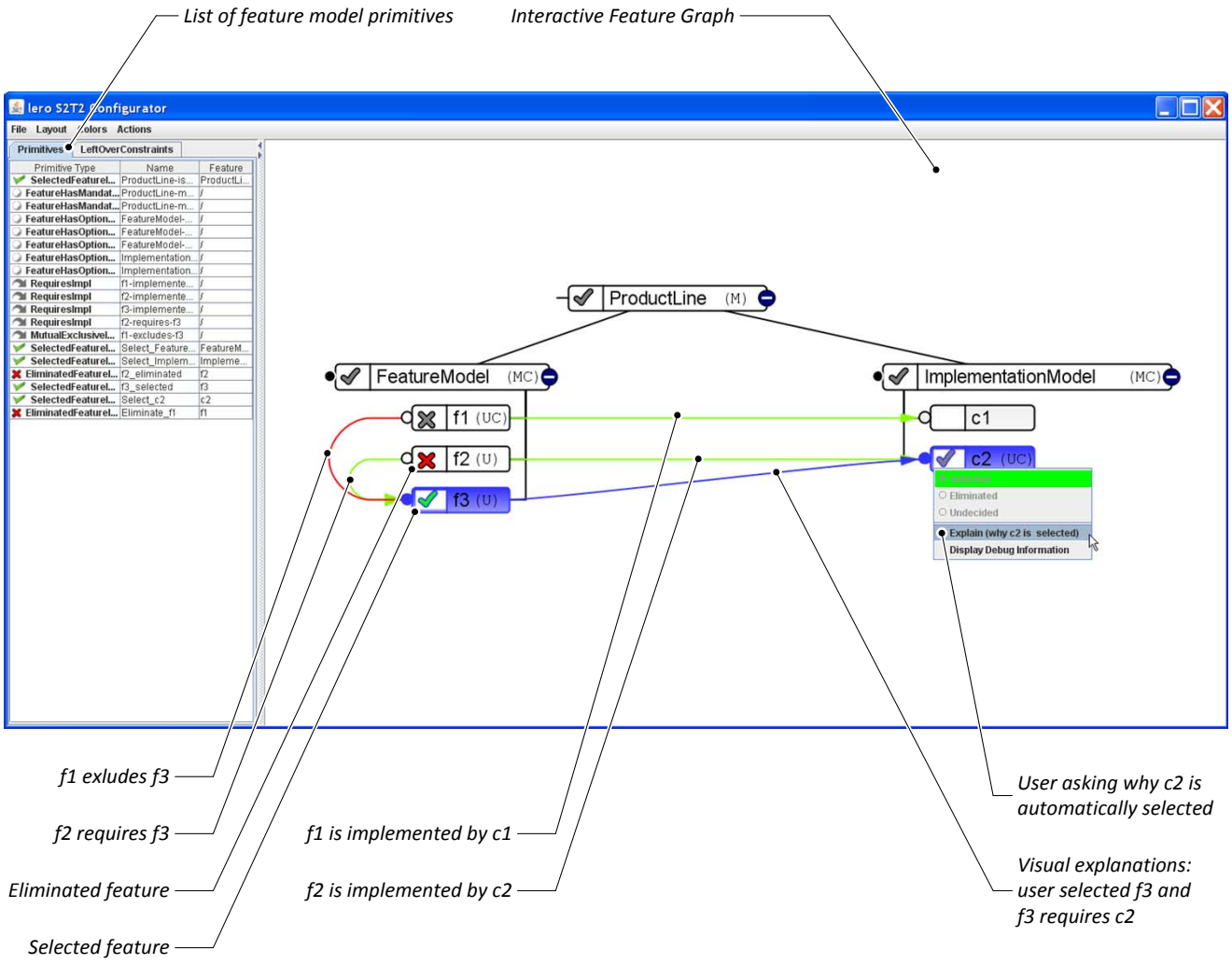

Fig. 3. The S2T2 Configurator showing an integrated SPL model during configuration.

their implementation, and relations between them in one model and (2) this model can be used in an interactive configuration tool.

V. Product Configuration

After converting the given SPL artefacts into one integrated model, we can use our tool `S2T2 Configurator` to perform an interactive configuration ❸.

Whenever a model is loaded, the Configurator internally transforms it into a formal representation, which is used by a reasoning engine to keep the configured model in an consistent state, to calculates consequences of the user's decisions and, on demand, and to provides visual explanations for such consequences [2].

Figure 3 shows the Configurator with a very simple model with just three features f_1 to f_3 (left-hand side) and two components c_1 and c_2 (right-hand side). Within the feature model, we have two dependencies (f_1 and f_3 are mutually exclusive; f_2 requires f_3). The features and components are connected via requires edges, which represent that features are implemented by certain components.

In the example, the user decided earlier that f_2 is eliminated and f_3 is selected (this is indicated by the red cross in front of f_2 and the green check mark in front of f_3). From these decisions and information in the model the tool derived that, f_1 has to be eliminated and c_2 has to be selected. In the screenshot, the user just asked why c_2 was selected (see the open context menu). The tool responded by highlighting f_3, c_2 and the requires edge between them.

We can apply this tool to handle more realistic models, which have been derived from a real Simulink implementation model (using the model transformation briefly explained earlier).

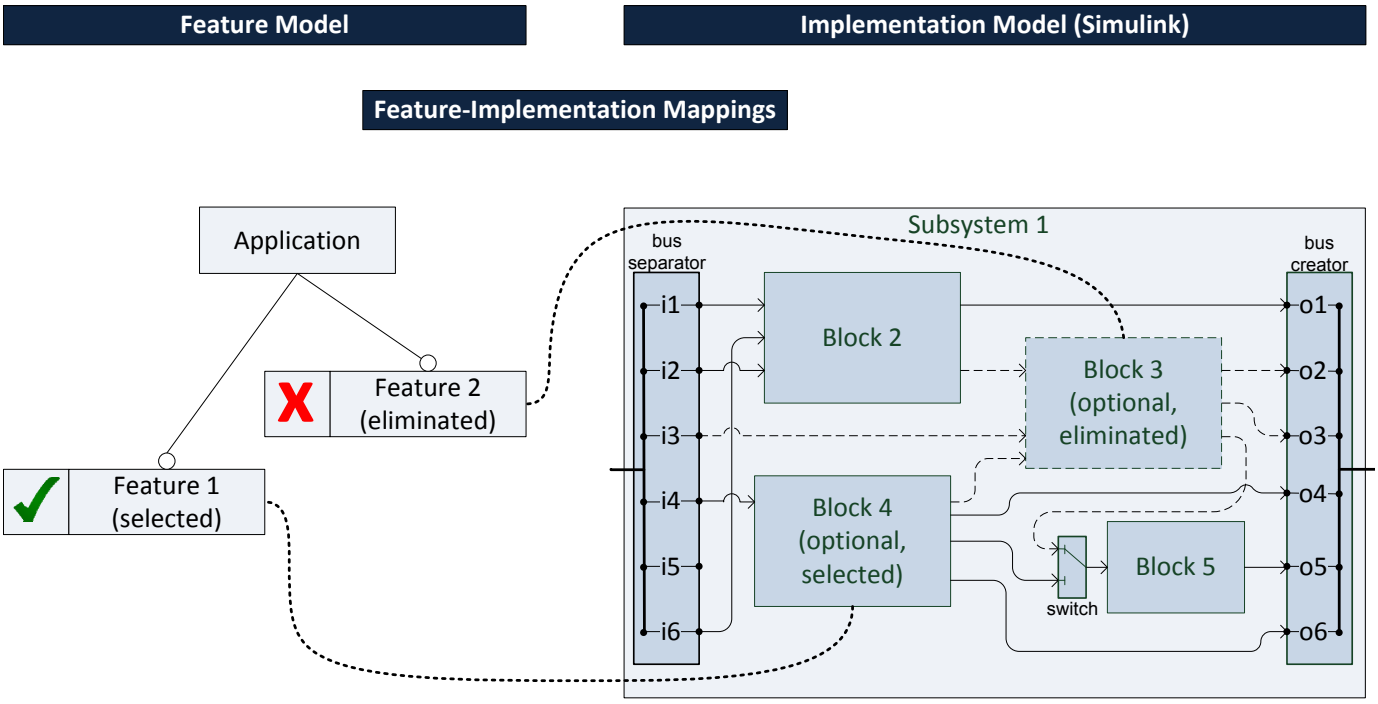

Fig. 4. Pruning of models.

VI. Product Derivation

Given the product configuration A_a we now have to turn this into an executable product. In the overview in figure 2 this corresponds to the process of *Product Derivation* ❹.

A. Negative Variability

The first step towards the executable product is the derivation of the Application Simulink Model C_a.

Here, we apply a well-known technique called negative variability: The Domain Simulink Model C_d contains the *union* of all possible product-specific Simulink Models C_a. Based on the configuration decisions in the product configuration A_a we then copy the Simulink models while filtering out all elements, which correspond to eliminated features. Hence, the term negative variability.

This technique can, for instance, be implemented with ATL model transformations (as demonstrated in earlier work [10]) or with openArchitectureWare's XVar component. For the approach discussed here we are currently experimenting with a connector that connects our Configurator to openArchitectureWare [8].

B. Pruning

The technique of negative variability is a first step, but it is not sufficient to get a consistent model. We will briefly discuss two situations, where we have to adapt more than just removing some affected blocks.

The first situation arises with alternative features. With such alternative groups of features, often the outputs of the corresponding implementation blocks are connected to the same port of a third block. This pattern is not a legal Simulink model, because Simulink does not know anything about the alternative group and the elements which will be removed later to obtain a legal model.

Hence, to create and test such a model within the Simulink tools, we have to introduce helper mechanism like *Switch* blocks. When we later apply negative variability, these helper mechanisms have to be adapted (or removed) as well. An example is depicted in the figure 4, where two signals lead into `Block 5` and are handled by a switch block. Whenever, only one of these two signals is left, we can remove the switch block altogether.

A second situation, where we have to adopt additional components in the Simulink model are *Bus* elements. These elements allow to combine multiple signals into one logical bus, to simplify the model (*Bus Creator*). In a different location in the same model, such a bus can again be separated into the single signals (*Bus Separator*). Whenever we apply negative variability, some signals within these buses might have to be removed (because the blocks providing these signals are no longer present). Hence, we have to adapt the bus. See the *bus creator* and *bus separator* in figure 4. In the given example, the signals i_3, o_2 and o_3 could be pruned.

VII. Discussion

In this work we implemented an approach, where we combined the feature, the mapping, and the implementation model into one big feature model. To this end, we transformed

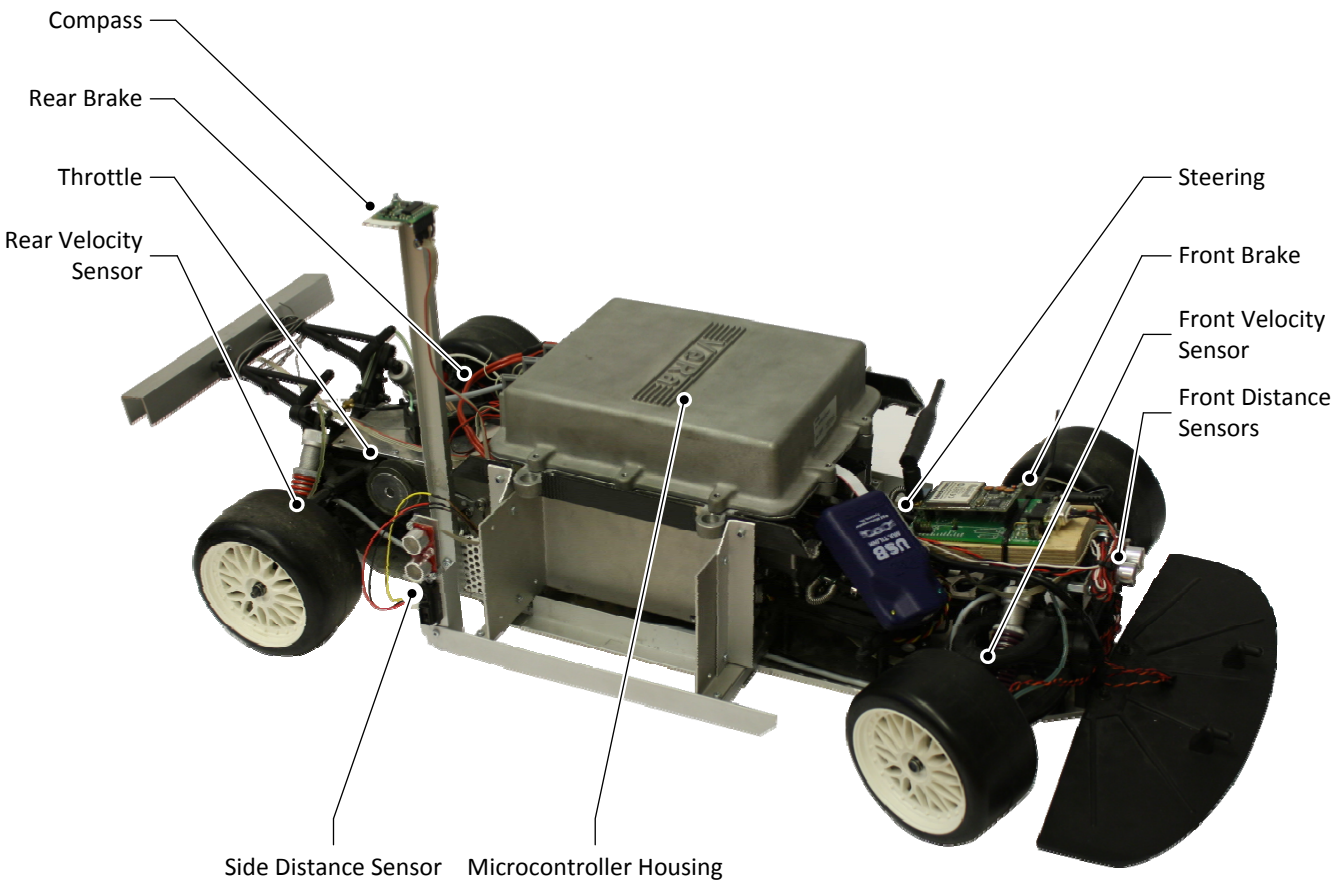

Fig. 5. The model car of the VeRa Rapid Control Prototyping platform.

the Simulink blocks of the implementation model into features and the mappings into constraints between the features and the blocks represented as features.

We experimented with this approach by using a product line of parking assistant applications, which is implemented using a Simulink model and a Rapid-Control-Prototyping system called VeRa. The model of the parking assistant can be simulate within Simulink or executed on a 1:5 model car, which is shown in figure 5. The application contains variability because of alternative distance sensors and an optional compass sensor, which helps the car to orientate itself in a parking bay.

This model contains a large number of blocks, subsystems and buses. Two parking algorithms are implemented to deal with the variability: One which uses the compass information and one without this information.

The transformation of a big Simulink model like our parking assistant into a feature model does not need remarkable time. So scalability in terms of execution time of the transformation seems to stay in reasonable bounds.

However, for larger models, during configuration the cognitive complexity and usability become an issue. Some techniques how to mitigate these problem with interaction techniques introduced to our `S2T2 Configurator` have been described in [3]. Up to now, we do not know if our approach scales in terms of usability. Hence, we intend to use large, realistic Simulink models and evaluate the configuration approach.

During the implementation we made the experience that Simulink is less strict about the syntax and the contents of values and parameters. This causes problems during transformation because the mechanisms further down the tool chain, such as Eclipse Modeling Framework (EMF, for handling models), ATLAS Transformation Language (ATL, for transforming models) and our Configurator are more strict about values.

For instance, it is perfectly legal to name a Block ”S-Function” in Simulink, actually including the quotes in the value. However, this will lead to technical problem when converting this to an EMF model. Similarly, Simulink does not care if names of blocks are unique. In EMF, however, it is desirable to have unique names since these are used as identifiers in references.

VIII. Related work

In earlier work we presented the basic architecture of the configurator [2] and discussed interaction techniques [3]. Here we extend this work by (1) a new approach of visualizing features and implementation, (2) using a configured feature model for product derivation and (3) pruning approaches to adapt the implementation model. The whole approach is

evaluated using a parking assistant as application and a tool chain using a Rapid Control Prototyping System.

Approaches which are related to our work can be roughly grouped in two categories, (1) approaches to deal with variability in domain-specific languages and (2) approaches to model variability in model-based development with Matlab/Simulink.

Weiland [11] addresses the challenges of variability in Matlab/Simulink. He uses marked standard Simulink blocks like switches to represent the different choices. Hence, the Simulink model contains the whole variability, a variant is then chosen by setting the corresponding parameters and selecting a specific signal path.

Kubica [12] starts from a feature model modeled in *pure::variants*, where the developer has to choose the desired features. Subsequently, the corresponding Simulink model is build automatically from templates and fragment models stored in the configuration tool.

There are other approaches, which are dealing with domain-specific techniques as well. For instance, Voelter and Groher [13] describe how to use openArchitectureWare [8] for Software Product Line Engineering. They use aspect-oriented and model-driven methods to generate products. To evaluate their approach they discuss a product line of Smart Home applications.

When dealing with variability, a typical challenge is the mapping of features or variation points to their implementation. Czarnecki and Antkiewicz [14] used a template-based approach where visibility conditions for model elements are described in OCL. Heidenreich et al. [15] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [16].

IX. Conclusions

In this paper, we presented an approach to the configuration of product lines within an existing tool for feature configuration.

The necessary translation from the implementation model into feature models and the targeted feature modeling language, present some limits with respect to expressive power. We can only "translate" model structures that are related to configuration, such as selection/elimination or $x-requires-y$ dependencies. More domain-specific concepts, e.g., voltages or oscilloscopes cannot be represented in a feature model in a meaningful way.

On the other hand, this translation enables us to configure an *integrated* model of the whole product line within one configuration tool. In particular, it provide the functionality described in the introduction:

1) The user can browse dependencies between features and the related elements in other models.
2) After applying configuration decisions he/she can immediately see consequences in related models.
3) It is possible to apply configuration decisions in the implementation model first and derive implications for the feature model from that.

In future work, we intend to improve (1) the product derivation mechanisms, including the connector which links our Configurator to openArchitectureWare and (2) the model transformation that implements the pruning.

X. Acknowledgments

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1 to Lero – The Irish Software Engineering Research Centre, http://www.lero.ie/.

References

[1] D. Beuche, "Modeling and building software product lines with pure::variants," in *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, September 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SPLC.2008.53

[2] G. Botterweck, M. Janota, and D. Schneeweiss, "A design of a configurable feature model configurator," in *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 09)*, January 2009. [Online]. Available: http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[3] G. Botterweck, D. Schneeweiss, and A. Pleuss, "Interactive techniques to support the configuration of complex feature models," in *1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009), held in conjunction with ECMDA 2009*, Twente, The Netherlands, June 2009. [Online]. Available: https://feasiple.de/public/proceedings-mdple2009.pdf

[4] J. Lunze, *Regelungstechnik 1*. Springer, 2004.

[5] A. Polzer, S. Kowalewski, and G. Botterweck, "Applying software product line techniques in model-based embedded systems engineering," in *6th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009)*, Vancouver, Canada, May 2009. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MOMPES.2009.5069132

[6] A. Polzer, G. Botterweck, and S. Kowalewski, "Variabilität im modellbasierten Engineering von eingebetteten Systemen," in *7. Workshop Automotive Software Engineering*, 2009.

[7] Eclipse-Foundation, "Eclipse Modeling Project." [Online]. Available: http://www.eclipse.org/modeling/

[8] "openArchitectureWare homepage." [Online]. Available: http://www.openarchitectureware.org/

[9] Eclipse-Foundation, "ATL (ATLAS Transformation Language)." [Online]. Available: http://www.eclipse.org/m2m/atl/

[10] 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)*, Atlanta, GA, USA, 2007, pp. 469–472. [Online]. Available: http://dx.doi.org/10.1145/1321631.1321711

[11] J. Weiland and E. Richter, "Konfigurationsmanagement variantenreicher simulink-modelle," in *Informatik 2005 - Informatik LIVE!, Band 2*. Koellen Druck+Verlag GmbH, Bonn, September 2005.

[12] S. Kubica, "Variantenmanagement modellbasierter funktionssoftware mit software-produktlinien," Ph.D. dissertation, Universität Erlangen-Nürnberg, Institut für Informatik, 2007, arbeitsberichte des Instituts für Informatik, Friedrich-Alexander-Universität Erlangen Nürnberg.

[13] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, September 2007. [Online]. Available: http://www.voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf

[14] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*, Tallinn, Estonia, September 29 - October 1 2005. [Online]. Available: http://dx.doi.org/10.1007/11561347_28

[15] 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*. New York, NY, USA: ACM, 2008, pp. 943–944. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370199

[16] Eclipse-Foundation, "EMF - Eclipse Modelling Framework." [Online]. Available: http://www.eclipse.org/modeling/emf/