# Using Higher-order Transformations to Derive Variability Mechanism for Embedded Systems

Goetz Botterweck[1], Andreas Polzer[2], and Stefan Kowalewski[2]

[1] Lero, University of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie
[2] Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
{polzer|kowalewski}@embedded.rwth-aachen.de

**Abstract.** One approach to handle the complexity of embedded systems is the use of models and domain-specific languages (DSLs) like Matlab / Simulink. If we want to apply such techniques to families of similar systems we have to describe their variability, i.e., commonalities and differences between the similar systems. Here, approaches from Software Product Lines (SPL) and variability modeling might be helpful. In this paper, we discuss three challenges which arise in this context: (1) We have to integrate mechanisms for describing variability into the DSL. (2) To efficiently derive products, we require techniques and tool-support that allow us to configure a particular product and resolve variability in the DSL. (3) When resolving variability, we have to take into account dependencies between elements, e.g., when removing Simulink blocks we have to remove the signals between these blocks as well. The approach presented here uses higher-order transformations (HOT), which derive the variability mechanisms (as a generated model transformation) from the meta-model of the DSL.

## 1 Introduction

Embedded systems are present in our everyday life. For instance, they are integrated into washing machines (to save energy and water), mobile devices (to simplify our lives) and in cars (to guarantee our safety).

In many cases, the engineering of embedded systems has to fulfill conflicting goals, such as reliability and safety on the one hand and the need for cost reductions and economic efficiency on the other hand. Moreover, the complexity of such systems is increasing due to the extension of functionality and increased communication with the environment. One possibility to deal with the complexity, requirements and the cost pressure is to use model-based development techniques like Matlab / Simulink. The advantages of such an approach are that connections between system components are expressed in an intuitive way on a higher abstraction level, which hides implementation details. Other benefits are support for simulation and increased reuse due to the component-oriented approach.

In the context of this paper we regard a “system” as a Matlab / Simulink model that contains components (Blocks) and provides a certain functionality. Nowadays, such systems are reused with small, but significant changes between different applications. Such variability causes additional complexity, which has to be handled. Some well-known techniques for this are suggested by Software Product Lines (SPL) [1,2]. In the research presented here, we discuss how these SPL techniques can be applied and adapted for the domain of model-based development of embedded systems.

The main contributions of this paper are (1) an analysis of Matlab / Simulink mechanism for variability, (2) concepts for managing realizing this variability with model transformations, (3) a mapping mechanism which adjusts the model according to configuration decisions (extension of [3]), and (4) concepts for "pruning", i.e., the cleanup of components that are indirectly influenced by configuration decisions.

The remainder of the paper is structured as follows: First, we will give an overview of our approach (in Section 2). After this we will explain methods of modeling variability with Matlab / Simulink (Section 3) and how the suggested variability concepts are managed (Section 4). Subsequently, we explain the additional pruning methods (Section 5) and the implementation with model transformations (Section 6).

## 2 Overview of the Approach

We address the challenges described in the introduction with a model-driven approach that relies on higher-order transformations. Before we go into more details, we will give a first overview of our approach (see Figure 1). The approach is structured into two levels (1) *Domain Engineering* and *Application Engineering*, similar to other SPL frameworks [2,1]. Please note that we mark processes with numbers (❶ to ❼) and artefacts with uppercase letters ($\boxed{A}$ and $\boxed{C}$, $\boxed{B}$ will be used in later figures). In addition, we use indexes (e.g., $\boxed{A_d}$ and $\boxed{A_a}$) to distinguish artefacts on Domain Engineering and Application Engineering level.

### 2.1 Domain Engineering

*Domain Engineering* starts with the consideration of the context and requirements of the product line during *Feature Analysis* ❶ leading to the creation of a *Domain Feature Model* $\boxed{A_d}$, which defines the scope and available configuration options of the product line. Subsequently, in *Feature Implementation* ❷ a corresponding implementation is created. Initially, this implementation is given in the native Simulink format (`*.mdl` files). To access this native implementation in our model-based approach, we have to convert it into a model. For this we use techniques based on Xtext [4]. (see [5,6] for more details). As a result we get the corresponding *Domain Implementation Model* $\boxed{C_d}$.

To prepare the derivation of products a higher-order transformation (HOT) ❸ is executed, which reads the DSL meta-model and generates the derivation transformation ❻, which will later be used during *Application Engineering*.

### 2.2 Application Engineering

The first step in *Application Engineering* is *Product Configuration* ❹, where, based on the constraints given by the *Domain Feature Model* $\boxed{A_d}$, configuration decisions are

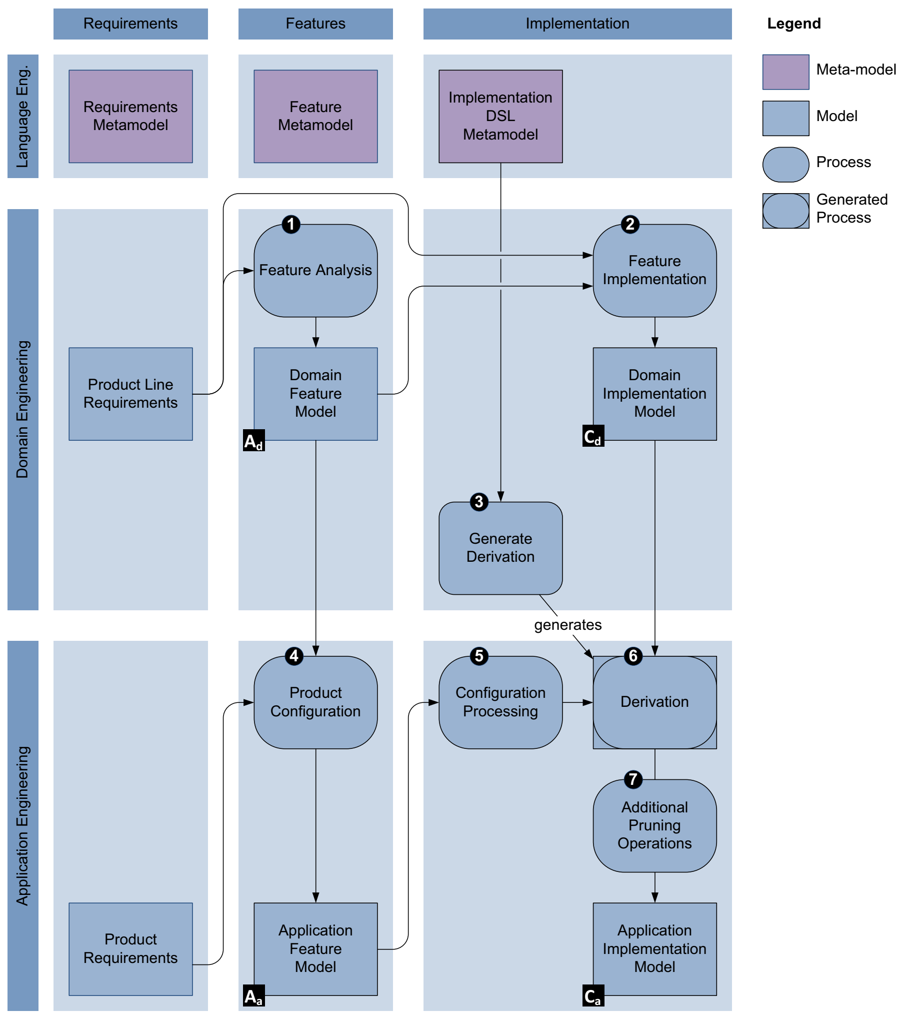

**Fig. 1.** Overview of the approach.

made, which defines the particular product in terms of selected or eliminated features. This results in a product-specific configuration which is saved in the *Application Feature Model* $\boxed{A_a}$. After some further processing ❺, this configuration is used in the *Product Derivation* ❻ transformation generated earlier by the HOT. This derivation reads the *Domain Implementation Model* $\boxed{C_d}$ and – based on the product-specific configuration – derives a product-specific implementation. After additional pruning operations ❼, the result is saved as the *Application Implementation Model* $\boxed{C_a}$, which can be used in further processing steps (e.g., Simulink code generation) to create the final executable product.

We will now describe these processes in more detail. We start with the process of *Feature Implementation* (❷ in Section 3). We will then explain the required adaptation

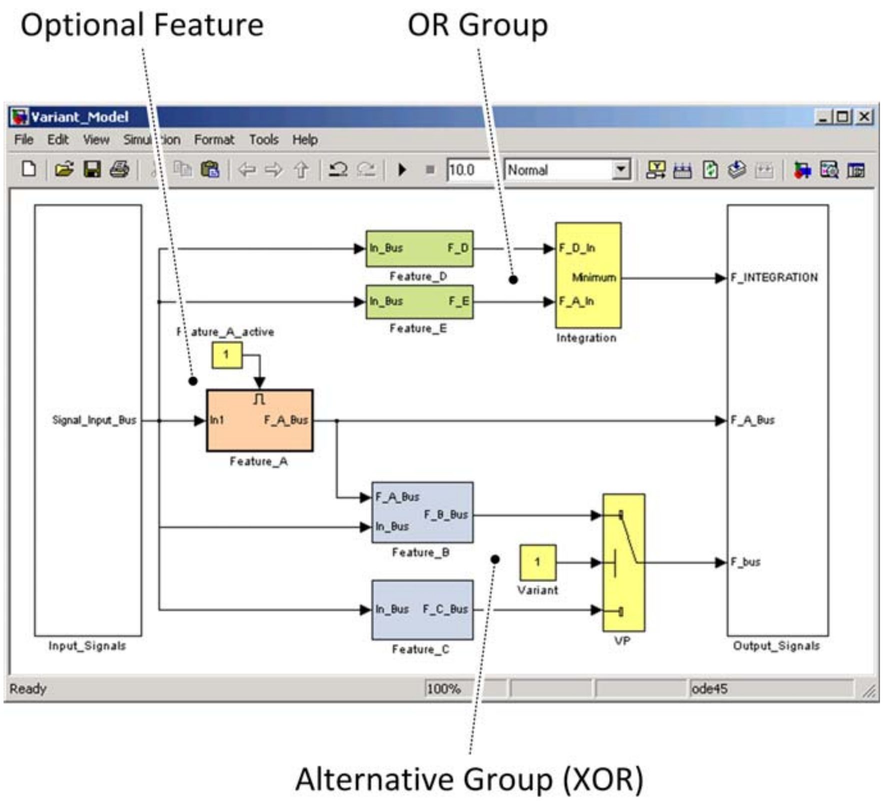

**Fig. 2.** Implementing Variability within a Matlab / Simulink model.

techniques including the *Derivation* ❻ (Section 4) and the subsequent *Pruning Operations* ❼ (Section 5).

## 3 Variability with Matlab / Simulink

Matlab / Simulink is a modeling and simulation tool provided by Mathworks. The tool provides *Blocks* which can be represented graphically and *Lines* which indicate communication. Blocks can contain other blocks, which allows to abstract functionality in special blocks called *Subsystems*. A similar technique is used by summarizing multiple lines into *Buses*.

In many cases, a feature can be implemented in Matlab / Simulink by a subsystem which contains the functionality for this particular feature. All necessary information consumed by the subsystem has to be provided by an interface, called *Input ports*. These input ports are normally buses. The information provided by a component are made available via *Output ports*, which are again organized buses and some additional signals.

We identified different possibilities to introduce variability in Matlab / Simulink for the required *Feature Implementation* ❷. Predominantly, we use the technique called *negative variability*. When applying this technique a model is created, which contains the *union* of all features across all possible configurations. When deriving a product-specific model this "union" model is modified such that features that were selected are activated and features that were not selected are deactivated or deleted.

To realize such a model in Simulink we can use two different approaches: (1) embedding the variability mechanisms *internally* (i.e., within the Simulink model) or (2) manipulating the model, based on variability information described *externally* (i.e., outside of the model).

We will now describe these two different techniques by explaining how common structures of feature models (Optional feature, Or Group, Alternative Group) can be implemented. For more information on these structures and feature models see [7,8]. An overview of feature diagram techniques and their formal semantics can be found in [9].

## 3.1 Variability mechanisms within the Simulink model

The first option is to realize variability by inserting artificial elements *into* the Simulink model. See the example model in Figure 2 where the blocks that implement features (*Feature_A*, *Feature_B*, *Feature_C*, ...) have been augmented with additional elements for variability (*Feature_A_active*, *Integration*, *Variant*, *VP*)

*Mandatory features* are – by definition – present in all variants. Hence, there is nothing to do for the implementation of mandatory features when deriving a product. There is no mandatory feature in the example.

*Optional features* can be realized in Matlab / Simulink models as a *triggered subsystem*, which is a special block that can be activated using a boolean signal (see *Feature A* in Figure 2). By using these mechanisms we are able to activate or deactivate a feature implementation.

When modelling *Alternative (XOR) group* and *Or group* we have to realize similar variability mechanisms. However, in addition we have to take care of the resulting signals and how they are fed into subsequent blocks. For alternative features we apply a Switch block (see the block *VP*in Figure 2) to choose the right output signal.

For *OR-related features* the integration of the results cannot be described for the general case, but has to be implemented depending on the particular case. In particular we have to take into the account the case when more than one feature of the group is selected and present in the implementation. We can implement an integration block for such cases in different ways. One example, is the case where limit is calculated for a certain dimension. Then the integration can be done by using a minimum (or maximum) function. In doing so, the lowest (highest) value is used by the system. As an example for this see *Feature D* and *Feature E* which are combined using an *Integration* block.

## 3.2 Variability mechanisms outside of the Simulink model

A second possibility, besides variability within the model, is the direct manipulation of the model with an *external* tool. To this end, during product derivation it is necessary to analyze the structure of a model file and delete the blocks representing deactivated features in such a way that a desired configuration is obtained.

When creating the union model it might be necessary to create a model, which could not be executed as-is within Simulink since the execution semantics of the created structure is undefined. This is because the variability decisions are still to be made and we have not yet removed certain elements.

As an example consider the case when we want to combine signals of two alternative features in an XOR group. In the union model, we might create the blocks for these two features side-by-side and feed their results into the same inport of the next block. As long as we configure this model (i.e., when the variability is applied), some parts will be removed, such that we get a valid and executable Simulink model. However, if we leave the union model (i.e., no variability applied) we do not realize the exclusion of the two features ("the X in XOR"). This leads to two signals feeding into on inport, which is an invalid Simulink model with undefined execution semantics.

In the example in figure Figure 2 this would correspond to connecting the *F_B_Bus* outport of Block *Feature_B* directly to the port *F_bus* of the bus *Output_Signals*, while at the same time connecting the *F_C_Bus* outport of Block *Feature_C* directly to the same port *F_bus*. If we would try to execute such a model, Simulink would respond with an error message.

The advantage of this kind of external method is that we do not have to pollute the domain model with artificial variability mechanisms.

Analyzing both possibilities, we came to the conclusion that a combination of both is an appropriate way of introducing variability. There are two major requirements which have to be fulfilled introducing variability methods. On the one hand we have to keep the characteristics of model based development (e.g., easy testing and simulation, capturing of dependencies possible), on the other hand the derived product should no longer contain any variability mechanisms. The mechanisms, which we introduced to realize this are explained in the next section.

## 4 Managing Variability

In this section we introduce the variability mechanisms we used in Simulink models and how they are influenced by configuring a corresponding feature tree. For instance the feature tree shown in Figure 3. This feature tree has an optional `Feature A`, XOR-grouped `Feature B` and `Feature C`, and OR-grouped `Feature D` and `Feature E`. Additionally the `requires` relation indicates that `Feature B` needs `Feature A`, i.e., whenever $B$ is selected $A$ has to be selected as well. This structure defines a set of legal configurations. Each of these configurations contains a list of selected features.

The mechanism that implements the structure of the feature tree has to fulfill certain requirements. In particular, it is important to keep the ability of simulating and testing the model. Therefore, it is necessary to build a model which has correct syntax, even after the variability mechanisms have been introduced. Additionally, the developer must have the possibility to introduce new features directly in the model. But due to the fact

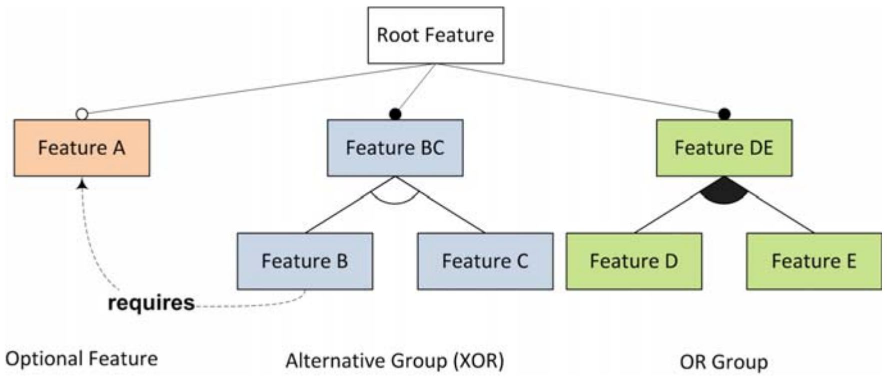

**Fig. 3.** Feature Tree for the variant Matlab / Simulink model shown in Figure 2.

that there are space and performance requirements the mechanisms have to be removed if the model is converted into a program executable on an embedded hardware.

To this end we used in general the approach of modeling variability *within* Simulink but with blocks which are specially marked as variability blocks. These special blocks are adopted afterwards according to the desired configuration. This means for obtaining a configuration, features which are not necessary will be deleted. Additionally signals between blocks are rerouted to be able to eliminate the variability blocks, which are not necessary in the derived product. The exact methods for a *switch*-block, *triggered subsystem* and, arbitrary integration mechanism is given in the following paragraphs.

The *switch*-block is used to express the relation between alternative grouped features. Therefore only one of them will give their contribution to the system. Using this, the developer of the implementation is able to simulate all features simply by choosing a configuration of the switch which selects to corresponding feature implementation block.

When deriving the executable product, it is necessary to delete those features that are not selected in the feature configuration. The output of each selected feature has to be connected with the port the switch block points to. All other corresponding signals have to be removed. In the end, no variability mechanism should be left in the model.

The situation is a bit simpler for triggered subsystems, which implement optional features. During simulation these blocks can be selected easily using *trigger*-signals. This will take effect on the simulation. If the corresponding feature is selected then the *trigger*-port has to be activated. When deriving an executable program, whenever the corresponding feature is deactivated, the subsystem has to be deleted. If it is activated nothing has to be done.

The mechanism to join the signals of OR-grouped features are not so easy to adopt. In the case of simulating the system the developer has to activate the desired features. This can be done either by using triggered subsystems to implement the features or just by disconnecting their signals with the block which joins the signals.

In case of deriving a real product two cases have to be distinguished. If only one feature is selected the other feature and the block joining the signals can be deleted. If more than one feature is selected the feature which are not selected can be deleted. But in this case the integration mechanism is still necessary.

## 5 Pruning

Dealing with blocks implementing features and the mechanisms to express variability is not the only adaptation which has to be done. There are additional components in the Simulink model which have to be changed as well, depending on the choices in the configuration

Especially, the interfaces for input or output signals which provide the information and supply the result from and to other systems have to be considered here. These blocks are interrelated with the configuration. They are necessary to define access to the signals. Therefore all signals provided from other systems are listed and treated in a way that the information is available in a common way. For instance the information for an input port is stored, renamed and mapped to internal representations which realizes the concrete representation.

Most of the signals, which are provided in the interface blocks are only needed for one feature. If this feature is not present the corresponding signals can be cleared out to optimize the implementation.

Buses that contains more than one signal have to be adopted in a similar fashion, i.e., only the required signals should remain within the buses. Hence, we have to prune out signals which are no longer needed because the corresponding features are to be removed.

These adoptions are not only necessary on the highest level of the model but also for subsystems. In general, it is possible that features are not visible at the highest level, for instance when subsystems are used to implement features. Since these subsystems use the same techniques as the main model of the whole system, it is necessary to adopt their interfaces and buses recursively in a similar fashion.

## 6 Implementation

The implementation discussed here (see Figure 4) is a technical realization of the approach shown earlier (see Figure 1).

The technical implementation follows the same structure, with Domain Engineering (Processes ❶ to ❸) and Application Engineering (processes ❹ to ❼). We will now discuss the model transformations in more detail.

### 6.1 Generating the derivation transformation

The higher-order transformation (HOT) `Metamodel2Derivation.atl` ❸ reads the meta-model of the DSL and generates a model transformation ❻, which is able to copy instances of this DSL.

Some excerpts of `Metamodel2Derivation.atl` are shown in Listing 1. The transformation is generated as an ATL `query` which concatenates the output as one

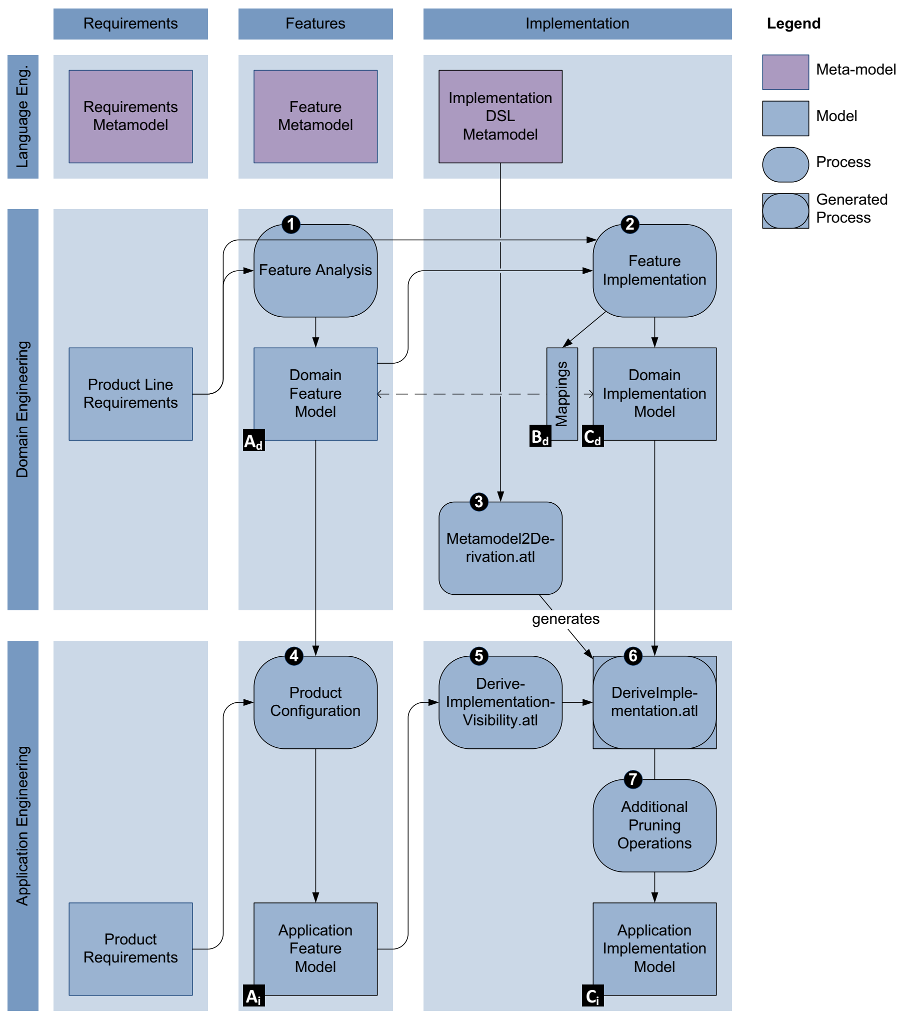

**Fig. 4.** Technical model workflow.

large string. For instance, the helper function `generateCopyRules()` (see Listing 1, lines 4–8) generates copy rules for all meta-classes in the meta-model. Details of each copy rule (e.g., the `rule`, `from` and to `part`) are generated by the function `Class.toRuleString()` (see Listing 1, lines 10–21) and other functions which were omitted for space reasons.

Examples of how the resulting transformation looks like, will be discussed in the next section.

```
query Metamodel2Derivation =
[..]

helper def: generateCopyRules() : String =
  ECORE!EClass
    ->allInstancesFrom('IN')
    ->sortedBy(o|o.cname())
    ->iterate(e; acc : String = '' | acc + e.toRuleString());

helper context ECORE!EClass def : toRuleString() : String =
  if not self."abstract" and self->inclusionCondition() then
    'rule ' + self->cname() + ' {\n' +
    '  from s : SOURCEMETA!' + self->cname() +
            self->inputConstraint() + '\n' +
    '  to t : TARGETMETA!' + self->cname() +
            ' mapsTo s (' +
            self->contentsToString() + ')\n' +
    '}\n\n'
  else
    ''
  endif;
[..]
```

**Listing 1.** Metamodel2Derivation.atl, transformation (3), excerpt.

## 6.2 Executing the derivation transformation

The generated derivation transformation `DeriveImplementation.atl` realizes a principle called "negative variability" [10] (also known as a "150% model"). With negative variability the domain model, here the *Domain Implementation Model* contains the *union* of all potential product-specific models.

Hence, the derivation transformation has to *selectively* copy the elements, filtering out those that should not be included, copying only those which will become part of the product-specific model, here the *Application Implementation Model* .

This selective copying is realized using the following mechanisms:

– For each meta-class in the DSL there is one copy rule. For instance, the rule `Block` (see Listing 2, lines 9–19) will copy instances of the meta-class `Block`.
– Each copy rule contains a condition that refers to an `.isVisible()` helper function, which controls whether an element is "visible" for the particular product and, hence, is copied or not. For instance, when processing the source element `s` the rule `Block` checks whether `s.isVisible()` (see Listing 2, line 12).
– To avoid inconsistent references, the following check is performed: Whenever references are processed, it is checked if the referenced elements are visible, as well. For instance, when copying references to `.generalization`, `.ownedAttribute`, and `.ownedOperation` the visibility is checked (see Listing 2, lines 16–18).

```
 1  module DeriveImplementation;
 2
 3  create TARGET : TARGETMETA from SOURCE : SOURCEMETA, CONFIG :
        CONFIGMETA;
 4
 5  helper context SOURCEMETA!Block def : isVisible() : Boolean =
 6    true;
 7  [..]
 8
 9  rule Block {
10      from s : SOURCEMETA!Block (
11          thisModule.inElements->includes(s) and
12          s.isVisible()
13      )
14      to t : TARGETMETA!Block mapsTo s (
15          name <- s.name,
16          subsystemblockbody <- s.subsystemblockbody->
                select(o|o.isVisible()),
17          normalblockbody <- s.normalblockbody->
                select(o|o.isVisible()),
18          scopeblockbody <- s.scopeblockbody->
                select(o|o.isVisible())
19    )
20  }
21  [..]
```

**Listing 2.** DeriveImplementation.atl, transformation (6), excerpt.

– The visibility functions determine whether instances of a certain meta-class will be copied. For instance, `Block.isVisible()` (see Listing 2, lines 5–6) calculates this for each instance of `Block`. In the initial version of `DeriveImplementation.atl` which is automatically generated from the meta-model all visibility functions default to true.
– In a second transformation, `DeriveImplementationVisibility.atl` ❺ Listing 3 these visibility functions are manually redefined. These functions access the product configuration and determine, which elements go into the product and which do not. Later on, these will be overloaded over the default visibility functions, by using ATL's superimpose mechanisms.

The selective copying is controlled by the function `s.isVisible()` defined in `DeriveImplementationVisibility.atl` ❺. This function reads the `Application Feature Model` $\boxed{A_s}$ and decides how this influences the filtering of elements in the `Domain Implementation Model`.

For this decision to be made, it is necessary to know how the various features in the feature model ($\boxed{A_d}$ and $\boxed{A_s}$) are related to the corresponding elements in the Matlab / Simulink implementation model.

```
module DeriveArchitectureDetermineVisibility;

create TARGET : TARGETMETA from SOURCE : SOURCEMETA, CONFIG :
    CONFIGMETA;

-- true if Block is referenced by a selected feature
helper context SOURCEMETA!Block def : isSelected() : Boolean =
[..]

-- true if Block is referenced by an eliminated feature
helper context SOURCEMETA!Block def : isDeselected() : Boolean =
[..]

helper context SOURCEMETA!Block def : isVisible() : Boolean =
  if self.isSelected() then
    if self.isDeselected() then
      true.debug('feature conflict for block' + self.name)
    else
      true
    endif
  else
    if self.isDeselected() then
      false
    else
      true -- default to visible
    endif
  endif;
[..]
```

**Listing 3.** DeriveImplementationVisibility.atl, transformation (5), excerpt.

This is represented by the *Mapping* $B_d$ between the *Domain Feature Model* $A_d$ and the *Domain Implementation Model* $C_d$ implemented as a model which contains as elements dependencies. These dependencies relate features and the corresponding implementation, with a link mechanisms available from the meta-model. Using this meta-model we are able to map a feature given in the configuration model to a block given in the implementation model.

To implement pruning operations we are currently experimenting with and implementing new user-defined methods. These methods will adopt the copy rules in such a way that the methods given in Section 5 are realized. These pruning operations are influenced by the configuration and the mapping of features. However, they affect model components which are only *indirectly* referenced by the mapping model.

## 7 Related Work

Several projects deal with Product Derivation. The ConIPF project provides a methodology for product derivation [11]. ConIPF concentrates on the formalization of derivation knowledge into a configuration model. Deelstra et al. provide a conceptual framework for product derivation [12].

When dealing with variability in domain-specific languages a typical challenge is the mapping of features to their implementations. Here, Czarnecki and Antkiewicz [13] used a template-based approach where visibility conditions for model elements are described in OCL. In earlier work [14,15], we used mapping models and model transformations in ATL [16] to implement similar mappings. Heidenreich et al. [17] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [18].

Voelter and Groher [10] used aspect-oriented and model-driven techniques to implement product lines. Their approach is based on variability mechanisms in openArchitectureWare [19] (e.g., XVar and XWeave) and demonstrated with a sample SPL of home automation applications.

In earlier work [20,5], the authors have experimented with other mechanisms for negative variability (pure::variants Simulink connector [21] and openArchitectureWare's Xvar mechanism [19]) to realize variability in Embedded Systems. The mechanisms were applied to microcontroller-based control systems and evaluated with a product line based on the Vemac Rapid Control Prototyping (RCP) system.

The approach presented here can be seen as an integration and extension of work from Weiland [22] and Kubica [23]. Both presented mechanisms to adopt Matlab / Simulink-models based on feature trees. Weiland implemented a tool which influences certain variability points in a Simulink model. However, variability mechanisms are not removed during variability resolution. The approach given by Kubica constructs a new Simulink model for a derived product.

Tisi et al. provide an literature review on higher-order transformations [24] including a classification of different types of HOT. Oldevik and Haugen [25] use higher-order transformations to implement variability in product lines. Wagelaar [26] reports on composition techniques for model transformations.

## 8 Conclusion

In this paper we presented an approach to introduce and adopt variability in a model-based domain specific language (Matlab / Simulink) for developing embedded systems.

With our approach we are able to simulate and test variable Simulink-models by introducing mechanisms to manage variability and additionally derive models which contains only the product specific components. This provides us with memory and computation time efficient models.

All model transformations were implemented in the ATLAS Transformation Language (ATL) [16]. The original version was developed with ATL 2.0. We are currently experimenting with ATL 3.0 and its improved support for higher-order transformations.

Using this technique we are able to reuse previous work which implements the transformation from a domain specific language and a abstract variability mechanism. This approach is expanded by new methods to decided whether a feature is active and new domain specific methods are needed to adopt the implementation model.

## 9 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/. The higher-order transformation (❸ in the overview) was inspired by an ATL case study by Dennis Wagelaar.

## References

1. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA (2002)
2. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering : Foundations, Principles, and Techniques. Springer, New York, NY (2005)
3. Beuche, D., Weiland, J.: Managing flexibility: Modeling binding-times in simulink. [27] 289–300
4. Eclipse-Foundation: Xtext http://www.eclipse.org/Xtext/.
5. Polzer, A., Botterweck, G., Wangerin, I., Kowalewski, S.: Variabilitt im modellbasierten engineering von eingebetteten systemen. In: 7. Workshop Automotive Software Engineering, collocated with Informatik 2009, Luebeck, Germany (September 2009)
6. Botterweck, G., Polzer, A., Kowalewski, S.: Interactive configuration of embedded systems product lines. In: International Workshop on Model-driven Approaches in Product Line Engineering (MAPLE 2009), colocated with the 12th International Software Product Line Conference (SPLC 2008). (2009)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, Reading, MA, USA (2000)
9. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International. (2006) 136–145
10. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan (September 2007)
11. Hotz, L., Wolter, K., Krebs, T., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J.: Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press (2006)
12. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. The Journal of Systems and Software **74** (2005) 173–194
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, Tallinn, Estonia (September 29 - October 1 2005)
14. Botterweck, G., Lee, K., Thiel, S.: Automating product derivation in software product line engineering. In: Proceedings of Software Engineering 2009 (SE09), Kaiserslautern, Germany (March 2009)

15. Botterweck, G., O'Brien, L., Thiel, S.: 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) 469–472
16. Eclipse-Foundation: ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/.
17. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: ICSE Companion '08: Companion of the 13th international conference on Software engineering, New York, NY, USA, ACM (2008) 943–944
18. Eclipse-Foundation: EMF - Eclipse Modelling Framework http://www.eclipse.org/modeling/emf/.
19. openarchitectureware.org: Official open architecture ware homepage http://www.openarchitectureware.org/.
20. Polzer, A., Kowalewski, S., Botterweck, G.: 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)
21. Pure::systems: pure::variants Connector for Simulink http://www.mathworks.com/products/connections/product_main.html?prod_id=732.
22. Weiland, J., Richter, E.: Konfigurationsmanagement variantenreicher simulink-modelle. In: Informatik 2005 - Informatik LIVE!, Band 2, Koellen Druck+Verlag GmbH, Bonn (September 2005)
23. Kubica, S.: Variantenmanagement modellbasierter Funktionssoftware mit Software-Produktlinien. PhD thesis, Universität Erlangen-Nürnberg, Institut für Informatik (2007) Arbeitsberichte des Instituts für Informatik, Friedrich-Alexander-Universität Erlangen Nürnberg.
24. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. [27] 18–33
25. Oldevik, J., Haugen, O.: Higher-order transformations for product lines. In: 11th International Software Product Line Conference (SPLC 2007), Washington, DC, USA, IEEE Computer Society (2007) 243–254
26. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: ICMT. Volume 5063 of Lecture Notes in Computer Science., Springer (2008) 152–167
27. Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings. In Paige, R.F., Hartman, A., Rensink, A., eds.: ECMDA-FA. Volume 5562 of Lecture Notes in Computer Science., Springer (2009)