Model-driven Support for Source Code Variability in Automotive Software Engineering

Cem Mengi*, Christian Fuß†, Ruben Zimmermann†, and Ismet Aktas‡

*Computer Science 3 (Software Engineering)
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
Email: mengi@i3.informatik.rwth-aachen.de

†Carmeq GmbH
Carnotstr. 4, 10587 Berlin, Germany
Email: {christian.fuss | ruben.zimmermann}@carmeq.com

‡Computer Science 4 (Distributed Systems)
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
Email: ismet.aktas@cs.rwth-aachen.de

***Abstract*—Variability on source code level in automotive software engineering is handled by C/C++ preprocessing directives. It provides fine-grained definition of variation points, but brings highly complex structures into the source code. The software gets more difficult to understand, to maintain and to integrate changes. Current approaches for modeling and managing variability on source code do not consider the specific requirements of the automotive domain. To close this gap, we propose a model-driven approach to support software engineers in handling source code variability and configuration of software variants. For this purpose, a variability model is developed that is linked with the source code. Using this approach, a software engineer can shift work steps to the variability model in order to model and manage variation points and implement their variants in the source code.**

***Index Terms*—automotive software engineering; programming; model-driven engineering; variability modeling;**

I. Introduction

Today the automotive industry provides customers a lot of possibilities to individualize their products. They can select from a huge set of optional fittings, e.g., parking assistant, rain sensor, intelligent light system, and/or comfort access system. The possibility to configure individual vehicles leads to the situation that both OEMs (Original Equipment Manufacturers) and suppliers have to capture explicitly potential *variation points* in their artifacts [1]–[3].

Thereby, the existence of variation points range over the whole electric/electronic (E/E) development process. They are available in the requirements, system specification, architecture design, source code, but also in the test and integration phase. Beyond that, variation points arise also during production, operation and maintenance phase. This means that in the whole product life cycle for a vehicle which hold up approx. 20-25 years, there evolve various types of variation points [4], [5]. Therefore, artifacts of different phases in the development process have to be investigated in order to explore their specifics [3], [6].

This paper deals with variability on source code level. Here, we focus on the programming languages C/C++, because they are the most widely used languages in automotive software engineering. With about 51% C has the most portion followed by C++ with about 30%. Assembler comes with about 8% and all other languages are applied less than 5% [7].

Variation points are implicitly modeled by implementing C/C++ preprocessing directives. In this way, variable (conditional) compilation results in specific software variants. This approach allows fine-grained definition of variation points, but brings highly complex structures into the source code. The software gets more difficult to understand, to maintain and to integrate changes. The main reason for this is that a software engineer has no support on source code level beside the programming language. Particularly, the user has to deal simultaneously with *problem space*, *configuration knowledge*, and *solution space* [8]. If a huge number of variation points exists, knowledge about a valid configuration gets difficult. Furthermore, a software developer has to find out the scattered code and the dependencies of one variant manually which is also very hard and time consuming.

There exists a wide range of techniques and mechanisms for modeling and managing variability [2], [9]–[15]. Most of them handle variability on a higher abstraction level. Elements of reusability are primarily software components, or constructs of object-oriented programming such as classes and methods which are replaced for specific variants of software. A support for fine-grained specifications of variation points on source code level are provided by a few number of concepts and tools [16]–[20], but they do not consider the specific requirements for the automotive domain. Particularly, safety critical applications come under regular code reviews and therefore have high demands on source code quality. Consequently, readability and understandability of source code are of high importance, but

the above mentioned existing solutions do not consider this sufficiently.

To close this gap, we propose a model-driven approach to support software developers in handling versatile source code and configuration of software variants. For this purpose, we have developed a concept to separate problem space, configuration knowledge, and solution space. The problem space includes a common cardinality-based feature model to capture and manage variability [10], [11]. Furthermore, it supports the possibility to configure a software variant. The configuration knowledge can subsequently be transformed to the solution space. The solution space contains the source code. Here, we use a view-based approach in order to display the current configuration and hide everything that do not belong to the configuration.

The paper is structured as follows: In Section II, we analyze preprocessing directives that can express variation points. Particularly, a detailed consideration will show where problem space, configuration knowledge, and solution space is integrated. Furthermore, the problems that we will treat will be described in detail by using an example. In Section III, we will describe our approach to solve the problems. Here, we explain the separation of problem space, configuration knowledge, and solution space and go into detail of the three parts. Section IV, contains a short description of our implementation approach. In Section V, we will check if we have solved the mentioned problems. Finally, Section VI will summarize the paper.

II. Analyzing Source Code Variability

In this section, we will investigate how variability can be expressed using C/C++ preprocessing directives. We will introduce an example in order to explain arising problems of this approach in more detail.

A. Expressing Variability with C/C++ Preprocessing Directives

The current approach to express variation points and to configure specific software variants is to apply C/C++ preprocessing directives. For this purpose, statements for conditional inclusions are used, e.g., **`#ifdef, #ifndef, #if, #elif, #else`** (see Figure 1) [21]. In the following, we will use *preprocessing block* or *block* as a synonym for complete preprocessing directives.

The `identifier` for **`#ifdef`** and **`#ifndef`** directives in Figure 1a and 1b is a point of variation, because depending on its evaluation the contained source code is either included for compilation or not.

In the same way, the `constant-expression` in **`#if`** and **`#elif`** preprocessing directives shown in Figure 1c and 1d is also a point of variation. If it is evaluated to nonzero, the appropriate part of source code is included for compilation, otherwise not. Note, that a `constant-expression` allows more complex arithmetic and logical expressions. In the following, we will use *block rule* or simply *rule* as a synonym for a constant expression.

```
#ifdef identifier
   ...
#endif
```

(a) **#ifdef** preprocessing directive.

```
#ifndef identifier
   ...
#endif
```

(b) **#ifndef** preprocessing directive.

```
#if constant-expression
   ...
#endif
```

(c) **#if** preprocessing directive.

```
#if constant-expression1
   ...
#elif constant-expression2
   ⋮
#elif constant-expressionN
   ...
#else
   ...
#endif
```

(d) **#if, #elif, #else** preprocessing directive.

Fig. 1. Preprocessing directives to handle variation points before compilation.

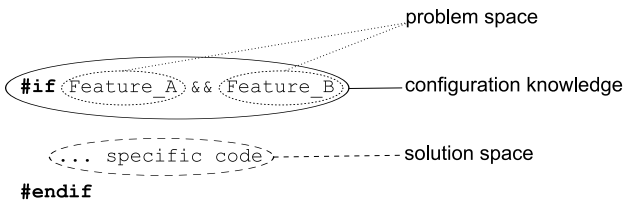


Fig. 2. Multilayer information in the solution space.

Analyzing preprocessing directives in detail, we have identified that different aspects of variability information is mixed into the code. We have decided to divide the information in analogy to Czarnecki's *generative domain model* which consists of a problem space, solution space and a configuration knowledge mapping between them [8]. Figure 2 illustrates this by an example.

The constant-expression `Feature_A && Feature_B` of the **`#if`** preprocessing directive is used to control the inclusion of the contained source code. Thereby, an identifier references a feature that is implemented in that code block, e.g., `Feature_A` and `Feature_B`. This kind of information is part of the problem space. The linking of an identifier with arithmetic and/or logical operations reflect configuration knowledge. Finally, the contained code reflect the implemen-

```
1  #if PRIO_USE_SORTED_OBJECTS == 1
2     #define PRIO_QUICKSORT 1
3     #define PRIO_INSERTIONSORT 0
4
5     ...
6
7     #if PRIO_QUICKSORT
8        ...
9     #endif
10
11    #if PRIO_INSERTIONSORT
12       ...
13    #endif
14
15    static void sortTracks(...) {
16       #if PRIO_QUICKSORT
17          quicksortTrack(...);
18       #elif PRIO_INSERTIONSORT
19          insertionsortTracks(...);
20       #else
21          #error missing ...
22       #endif
23
24    }
25 #endif
```

Fig. 3. An example for variability handling with preprocessing directives.

tation which is part of the solution space.

B. Problem Description by Example

In this section, we will explain the problems that currently exists when dealing with C/C++ preprocessing directives to handle variability information. For this purpose, we will introduce an example.

Typically, sensors are adopted to collect data. In some situations it is necessary to prioritize the captured data. If so, different variants of sorting algorithms can be applied, e.g., quick-sort or insertion-sort.

The associated C source code is shown in Figure 3. The code between line 1 to 25 is only included if prioritization is selected. One of the sorting algorithms have to be configured (set to 1) in order to integrate the appropriate source code into the software variant. In our case, it is the quick-sort (see line 2). Particularly, the `sortTracks(...)` function (line 15) includes only the part of the source code which belongs to the quick-sort algorithm (line 17).

Although using preprocessing directives allows fine-grained and flexible specification of variation points, the source code gets more difficult to understand, to maintain, and to integrate changes. Analyzing the source code, we have identified four main problems, i.e.,

1) mixing problem space, configuration knowledge and solution space,
2) viewing all variation points without the knowledge of a valid configuration,

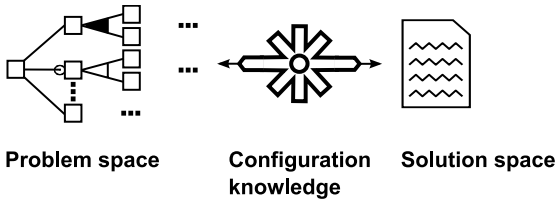


Fig. 4. Separation of problem space, configuration knowledge, and solution space.

3) code-variants of one variation point are scattered and have to be find manually, and
4) no explicit capturing of dependencies between variation points.

As described in Section II-A we have detected information in the source code that belongs to both problem space and solution space. For example, line 1, 7, 11, 16 etc. in Figure 3 are variability information that are part of the problem space and configuration knowledge. Even so, they are strongly integrated into the solution space, i.e., the source code.

Furthermore, considering the source code example, a software engineer always has to work with all variation points simultaneously, even most of them are not part of a specific variant. For example, the insertion-sort algorithm in Figure 3 does not belong to the variant if quick-sort is chosen (lines 3, 11-14, and 18-19). If more complex code sizes are regarded, solving a valid configuration gets more difficult.

Moreover, code-variants of one variation point are typically not implemented in a complete block but rather are scattered. For example, the quick-sort variant in Figure 3 appears in lines 2, 7-9, and 16-17. Particularly, this complicate including changes into code-variants or their appropriate preprocessing directives. If the code gets more complex, finding the code-variants manually gets very hard and time consuming. If changes into code or conditions have to be done, all relevant source code have to be find out manually to hold them consistent.

Finally, in many cases variation points are not isolated but depend on each other. In the source code, there is no explicit capturing of such information. For example, quick-sort and insertion-sort in Figure 3 are only included if a prioritization is necessary. If so, then they have an exclusive dependency on each other, i.e., only one can be chosen.

III. Model-driven Support for Source Code Variability

To deal with the identified problems mentioned in Section II-B, we propose a model-driven approach to treat source code variability and to support configuration of software variants. Therefore, we have developed a concept to separate problem space and configuration knowledge from solution space. The problem space is supported by a variability model that is based on Czarnecki's cardinality-based feature model [10], [11] (in the following we will use the term *variability model* as a synonym). Here, variation points are captured and

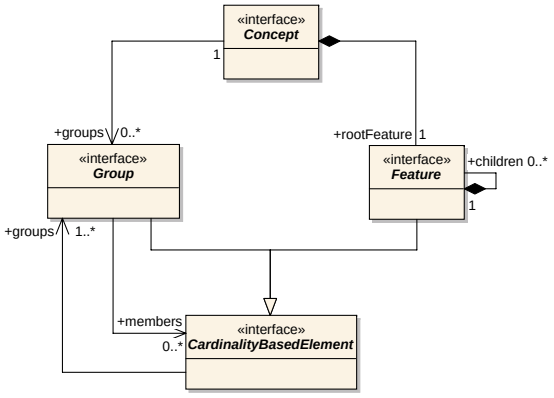


Fig. 5. Meta-model of the cardinality-based feature model.

managed. The configuration knowledge contains all informations to transform knowledge from problem space to solution space. The variability model supports the configuration. The solution space includes the source code. By integrating a view-based approach, only the configured part of the source code is displayed. This reflects the result from problem space transformed to solution space.

Figure 4 gives an overview of the separation of our approach. The general idea is, that a software developer not only work on the solution space, i.e., the source code, but also shift work steps into the variability model that is able to capture the problem space and configuration knowledge.

A. Source Code Variability Model

The focus on this paper does not lie on defining a new variability model, but rather using existing solutions to support variability on source code level. Analyzing existing approaches we have decided to adapt a cardinality-based feature model. Since it is a very common way to model variability, an integration of other tools and models get more simple. Particularly, this integration would allow using variability modeling techniques which are applied on a more abstract level, i.e., managing variability for classes, methods, objects etc. Our approach can then be used for fine-grained modeling of variability, i.e., on source code lines.

Figure 5 shows the meta-model for the cardinality-based feature model. It allows to define a tree-based structure. Thereby, a *Concept* node contains exactly one feature, i.e., the *rootFeature*. A Feature consists of an arbitrary number of *children* features. Moreover, a *Concept* node references an arbitrary number of *Groups* which define the number of elements in a group that can be specified for a configuration.

B. Transformation of Configuration Knowledge

To profit from the separation, it is an essential part to shift work steps to the central variability model. For this purpose, a connection between variability model and source code is necessary. To achieve this, we will use preprocessing directives which were, as described in Section II, the primary concept to express variation points. In this way, it will be possible to automatically add or delete preprocessing directives.

A user configures a specific variant whereas every modification of the source code, i.e., adding, deleting or modifying code lines, is linked with that configuration. Later on, it will be possible to display or hide code blocks depending on a specified configuration.

The basic principle for every transformation is the configuration knowledge from the variability model. If features *F_1, F_2, ..., F_n* are selected, a transformation into a rule of the form `F_1 && F_2 && ...&& F_n` is executed. In the following examples, we always assume that this constant-expression is used.

Depending on the modification of the source code, the constant-expression is integrated into a preprocessing directive.

1) Modification of Source Code Outside Existing Preprocessing Blocks: The most simple case is when a software engineer modifies code outside existing preprocessing blocks.

a) Adding: If we have a rule of the form `F_1 && F_2 && ...&& F_n`, then it is embedded to an **`#if`** preprocessing block:

```
#if F_1 && F_2 && ...&& F_n
   ...
#endif
```

In this way, the code is only included for compilation, if the appropriate configuration is selected.

b) Deleting: Deleting code lines during a given configuration *F_1, F_2, ..., F_n* do not delete them from the source file, but implies that the deleted lines should not appear in that configuration. For this purpose, we use the following **`#if`** preprocessing directive with the deleted code lines:

```
#if !(F_1 && F_2 && ...&& F_n)
   ... deleted code lines
#endif
```

2) Modification of Source Code Inside Existing Preprocessing Blocks: A slightly different case arises, if modifications inside existing preprocessing blocks are made.

a) Adding: If code is added inside a preprocessing block, then it is split in two blocks with the constant-expression as before and the added code lines are embraced with an **`#if`** preprocessing directive and a rule `F_1 && F_2 && ...&& F_n` that is transformed from the specified configuration.

```
#if constant-expression_1
   line_1
   line_2
   line_3
#endif
```

⇒

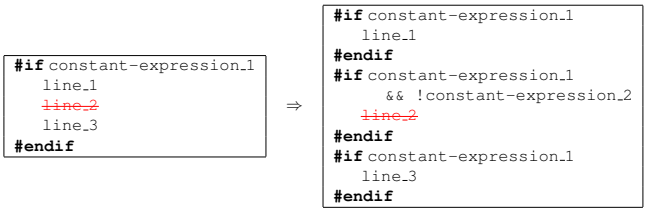
```
#if constant-expression_1
   line_1
#endif
#if constant-expression_2
   line_2
#endif
#if constant-expression_1
   line_3
#endif
```

In the example above, the blue marked code line (`line_2`) on the left side is added during a configuration *F_1, F_2, ..., F_n*. In that case, `line_1` and `line_3` are embraced with the preprocessing directive as before and `line_2` is embraced with an **`#if`** preprocessing directive and a constant-

expression F_1 && F_2 && ...&& F_n (in the figure above, denoted as constant-expression_2).

This adaptation differs from the transformation for modification of source code outside existing preprocessing directives. If we would transform the added code lines in the same way as in Section III-B1a then we would get a nested structure. But this would bring an implication into the code that possibly is not planned by a software developer. For example, if line_2 would be nested into the superior preprocessing block, then the code lines would only exist in a variant that includes a configuration of the superior block. By dividing them in multiple blocks of preprocessing directives this side effect is avoided and the described implication is still possible if the software engineer uses the configuration of the variability model.

b) Deleting: When deleting code lines, the mentioned problems for adding code do not appear, because the reference to the superior preprocessing block is mandatory and must be kept, so that the constant-expression of the superior preprocessing block and the constant-expression that is transformed from the current configuration must be included.

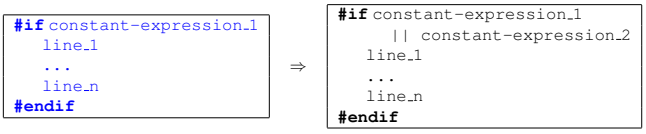


In the example above, the red marked and struck out code line (line_2) on the left side is deleted during a configuration *F_1, F_2, ..., F_n*. In that case, line_1 and line_3 are embraced with the preprocessing block as before and line_2 is included into an **#if** preprocessing block with a constant-expression constant-expression_1 && !constant-expression_2, where constant-expression_2 is the result of the transformation of the current configuration, i.e., F_1 && F_2 && ...&& F_n. We have decided to split the preprocessing block but nesting them would in this case also be possible.

If only **#if** !constant-expression_2 would be included then the deleted code line would be appear in each variant that do not contain the configuration *F_1, F_2, ..., F_n*. Particularly, it would be independent from constant-expression_1.

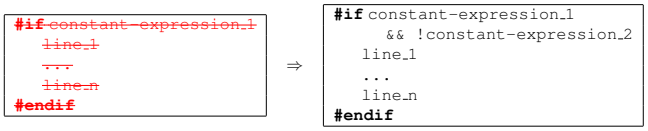
3) Modification of Source Code for Complete Preprocessing Blocks: Beside of adding or deleting code lines, in some situations it is also reasonable to add or delete complete preprocessing blocks in a given configuration.

a) Adding: If it is necessary to add a preprocessing block of one variant (or configuration) into another one, this could be done by configuring the variant where the code block appears, copying it, configuring the variant where it should appear, and then pasting it. This method is a little bit uncomfortable. Therefore, we support adding complete code blocks into a configuration automatically without copy/paste actions. For this purpose, we only have to extend the constant-expression of the preprocessing block which include the code lines that should appear in the current configuration.



In the example above, the blue marked code lines on the left side should included into a configuration *F_1, F_2, ..., F_n*. The appropriate preprocessing block after transformation is shown on the right side. The code line now would appear in configuration that is transformed to constant-expression_1 or constant-expression_2, where constant-expression_2 is the current configuration *F_1, F_2, ..., F_n*.

b) Deleting: If a complete preprocessing block is deleted, an adaptation of the constant-expression should be made. Thereby, the transformation of a configuration is motivated by the same principle as for deleting code lines inside existing preprocessing blocks.



In the example above, the red marked and struck out code lines on the left side are deleted during a configuration *F_1, F_2, ..., F_n*. The result of the transformation with the given configuration is shown on the right side. Considering the constant-expression on the right side, the appropriate code lines only appear if constant-expression_1 but not constant-expression_2 holds, where constant-expression_2 is equal to F_1 && F_2 && ...&& F_n.

C. Views on Source Code

The solution space of our approach is the source code. In order to take the advantages of the division the transformation of configuration knowledge must be reflect in the source code. For this purpose, we have adopt a view-based approach where all source code is hidden that is not part of the configuration.

The configuration is made on the cardinality-based feature model which was explained in Section III-A. Depending on the configuration, all constant-expressions of preprocessing directives are evaluated to decide whether the block should be displayed or hidden. In principle, this emulates the preprocessor with the advantage that targeted configurations can be viewed.

IV. Implementation

The described concepts are implemented in a way that they can be integrated into existing development processes and projects as seamless as possible. Therefore, we had to follow general requirements:

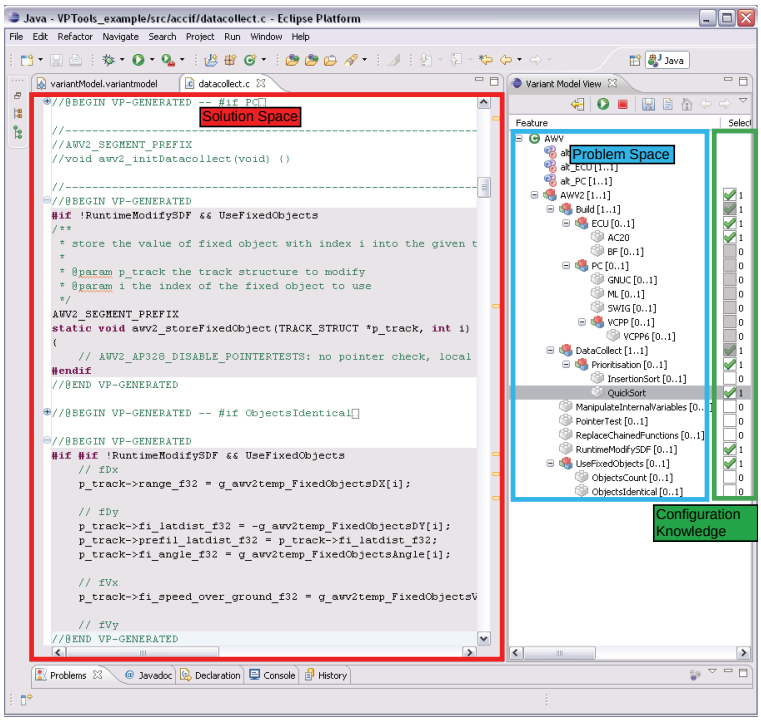


Fig. 6. A screenshot of the developed Eclipse plugin.

1) At all time, valid C/C++ code must be available.
2) Editing and maintenance of source code must be possible without the need for specific tooling.
3) Additional work load for a software developer must be as low as possible.
4) Dynamic changes must be feasible.

The implementation is fulfilled by developing a plugin for the *Eclipse Framework* [22]. The cardinality-based feature model is implemented with support of the *Eclipse Modeling Framework (EMF)* [23]. An editor for the feature model was also generated by using EMF which can be used in parallel to the Eclipse C/C++ Development Tooling (CDT) [24]. A screenshot is shown in Figure 6.

The left part contains the editor where C/C++ source code can be written. The right part contains a view on a configurable feature model. The software developer can use both parts in parallel in order to configure a specific variant of interest so that all other code lines that are not included into the variant are hidden. In some situations, not all modifications on model configuration should influence the view on the source code. In the same way, not all modifications on source code should influence the selected configuration. For this purpose, the user gets the possibility to explicitly select a control element that triggers the linking between code and model. If the linking is stopped the transformation is subsequently executed. This means, that all preprocessing directives are added into the source code.

The editor to configure a variant has the ability to select or deselect features and to solve implications. Furthermore, the configuration of invalid variants are avoided. At the same time, it supports a software developer to detect modeling errors.

V. Problems Revisted

If we consider again the listed problems in Section II-B, we observe that they are solved by the described concepts.

The core problem was that problem space, configuration knowledge, and solution space were mixed. By dividing them we have formed a basis to solve the other problems. Knowledge about a valid configuration is given through support of the configurable feature model. Code-variants must not find out manually, but are solved by the configuration which then is transformed to the source code. By adopting a view-based approach only the relevant code lines are displayed. Dependencies between variation points are also stored in the feature model by expressing cardinalities.

Overall, complex work steps are now shifted to a model where they can be handled more easier. The software engineer can now concentrate on the main work, i.e., developing software.

VI. Conclusion

In this paper we have described a model-driven approach to handle source code variability. We have outlined existing problems, analyzed them in detail in order to propose a solution. The main problem is that problem space, configuration knowledge, and solution space is mixed, i.e., a software engineer works only on the source code without any support to treat variability. This leads to the situation that source code is overcrowded with variation points without knowing how they depend on each other. In our approach we have suggest

a division of problem space, configuration knowledge, and solution space. A cardinality-based feature model is adopted and linked with the source code in order to shift work steps into the model. By a configuration support modifications on the source code are linked with the model. Furthermore, transformation of configuration is supported by adopting a view-based approach.

In future work, we want to integrate this approach with earlier phases of an E/E development process. Software architectures are one essential artifact that need support for variability handling. If variability support is provided, an integration with the source code level would be an essential benefit.

REFERENCES

[1] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, September 2005.

[3] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[4] M. Broy, "Challenges in automotive software engineering," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 33–42.

[5] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–71.

[6] C. Mengi and I. Armaç, "Functional Variant Modeling for Adaptable Functional Networks," in *VaMoS 2009: Third International Workshop on Variability Modelling of Software-Intensive Systems*, ser. ICB Research Report, vol. 29. Universität Duisburg-Essen, 2009, pp. 83–92.

[7] Embedded Systems Design website, http://www.embedded.com/.

[8] K. Czarnecki and U. Eisenecker, *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

[10] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[11] K. Czarnecki and C. H. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," in *OOPSLA'05 International Workshop on Software Factories*, October 2005.

[12] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families," in *SPLC 2004: Software Product Lines, Third International Conference*, ser. Lecture Notes in Computer Science, vol. 3154. Springer, 2004, pp. 197–213.

[13] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Sci. Comput. Program.*, vol. 53, no. 3, pp. 333–352, 2004.

[14] Pure Systems website, http://www.pure-systems.com/.

[15] T. Asikainen, T. Soininen, and T. Männistö, "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families," in *PFE 2003: Software Product-Family Engineering, 5th International Workshop*, ser. Lecture Notes in Computer Science, vol. 3014. Springer, 2003, pp. 225–249.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 311–320.

[17] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 162–180, 2008.

[18] C. Lopes and G. Kiczales, "Aspect-oriented programming," *Technology of Object-Oriented Languages, International Conference on*, vol. 0, p. 468, 2000.

[19] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy, "Explicit programming," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002, pp. 10–18.

[20] S. J. Paul, P. Bassett, H. Zhang, and W. Zhang, "Xvcl: Xml-based variant configuration language," in *ICSE '03: Proceedings of the international conference on Software engineering*, 2003, pp. 810–811.

[21] B. W. Kernighan and D. M. Ritchie, *C Programming Language, 2nd Ed*. Prentice Hall, January 1988.

[22] The Eclipse Foundation website, http://www.eclipse.org/.

[23] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[24] Eclipse C/C++ Development Tooling Project, http://www.eclipse.org/cdt/.