MANCHESTER 1824
The University of Manchester
The University of Manchester Research

A Component Model for Defining Software Product Families with Explicit Variation Points

DOI:
10.1109/CBSE.2016.15

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Di Cola, S., Tran, C., Lau, K.-K., Qian, C., & Schulze, M. (2016). A Component Model for Defining Software Product Families with Explicit Variation Points. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE): IEEE Computer Society Digital Library* https://doi.org/10.1109/CBSE.2016.15

Published in:
2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

OPEN ACCESS

Download date:14. Apr. 2024

A Component Model for Defining Software Product Families with Explicit Variation Points

Simone Di Cola, Cuong Tran, Kung-Kiu Lau, Chen Qian
School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
dicolas,ctran,kung-kiu,cq@cs.manchester.ac.uk

Michael Schulze
Pure-systems GmbH
Otto-von-Guericke Strasse 28
39104 Magdeburg, Germany
michael.schulze@puresystems.com

***Abstract*—In software product line engineering, the construction of an ADL architecture for a product family is still an outstanding engineering challenge. An ADL architecture for a product family would define the architectures for all the products in the family, allowing engineers to reason at a higher level of abstraction. In this paper, we outline a component model that can be used to define architectures for product families, by incorporating explicit variation points.**

***Keywords*-component model; product family architecture; explicit variation points;**

I. Introduction

Fig. 1 shows the key artefacts involved in the construction of product families in Software Product Line Engineering (SPLE) [34], [29]: feature model, architecture and components. The *feature model* [7] captures common and

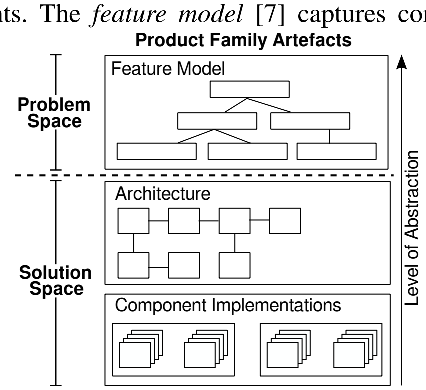

Figure 1: Product family artefacts [32].

variable characteristics in the *problem space* as nodes in a tree. Variability is expressed by *optional*, *alternative* and *or* variation points. The feature model is the most abstract specification of a product family. In order to realise the product family defined by a feature model, SPLE makes use of two kinds of artefacts in the *solution space*: an *architecture* for the product family; and *components* that can be combined into a product. However, the construction of an architecture in the sense of ADL (architecture description language) [26] for a product family is still an outstanding engineering challenge [17].

In this paper we outline a component model [23], called FX-MAN, that can be used to construct a real architecture for a product family, and thereby provide this crucial solution space artefact. We have implemented a tool for our model [14], and we demonstrate its use in SPLE on an example.

II. The FX-MAN Component Model

A component model for constructing product families must define a family of architectures by incorporating variation points, as well as composition mechanisms for combining (sub)families of architectures into larger ones.

The basic idea of FX-MAN is that it defines: (i) basic component-based architectures that correspond to features; (ii) variations of sets of basic architectures; (iii) composition of sets of basic architectures into a product family.

Basic component-based architectures are *X-MAN architectures*, constructed using the X-MAN component model [21], [25]. These are intended to implement features in the final products.

A set of X-MAN architectures is a family of product parts. We call such a set an *X-MAN set*. Variations of X-MAN sets are constructed by *variation operators* that correspond to standard variation points in feature models, namely *OPT* (optional), *ALT* (alternative, or exclusive *or*), and *OR* (inclusive *or*).

Tuples of X-MAN sets that represent variations generated by variation operators can be composed into a product family. Such a family contains all the possible products (containing all possible variations as defined in the feature model).

A. X-MAN Component Model

In X-MAN there are two kinds of components: (i) *atomic* and (ii) *composite* components. An atomic component consists of a *computation unit* (CU) and an *invocation connector* (IC). The computation unit contains the implementation of the services exposed by the invocation connector. Atomic components can be composed by *composition connectors*

Research leading to these results has received funding from the EU ARTEMIS Joint Undertaking under grant agreement no. 621429 (project EMC2) and from the Technology Transfer Board (TSB) on behalf of the Department for Business, Innovation & Skills, UK.

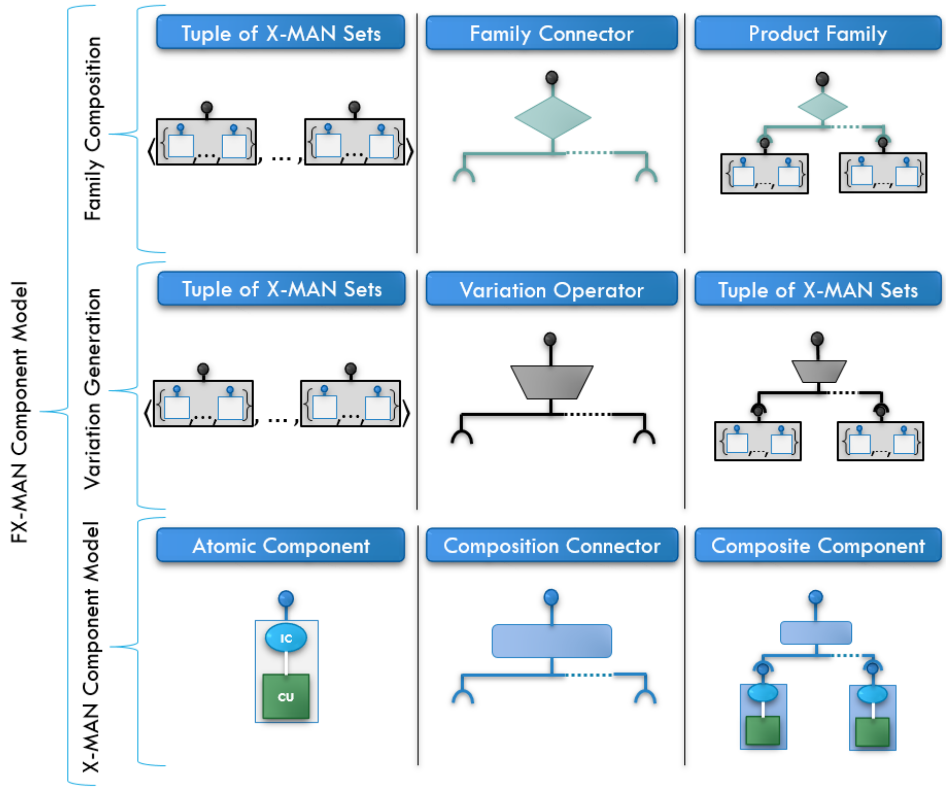

Figure 2: FX-MAN overview.

into *composite components*. Composition connectors are (exogenous) control structures that coordinate the execution of the components they compose [24]. A *sequencer* (SEQ) provides sequencing, while a *selector* (SEL) branching. AN *aggregator* connector (AGG) aggregates the services exposed by its sub-components.

B. Variation Generation

To generate variations of X-MAN sets, we have defined three *variation operators*, which are functions that take a tuple of X-MAN sets as input and return variations of the input sets. The resulting variations are again tuples of X-MAN sets.

A variation operator is a function that applies the variability expressed in a feature model, that is exclusive or (`ALT`), optional (`OPT`), and inclusive or (`OR`) to a tuple of X-MAN sets. The language of our variation operators is defined by a context free grammar.

The `ALT` variation operator is a function that takes a tuple of at least two T's as input, and returns each input set as a possible alternative. The `OR` variation operator also takes as input a tuple of at least two Ts, and returns all possible combinations (without repetition) of its input. The `OPT` operator makes a single T optional.

Variation operators can be nested, since they all return tuples of X-MAN sets. This is in keeping with the hierarchical nature of variation points in a feature model.

C. Family Composition

Once variations of X-MAN sets have been generated, the X-MAN architectures in these sets can be composed together into a family of products, which is another tuple of one X-MAN set. The composition of these sets can be defined in terms of X-MAN composition connectors, since it is ultimately X-MAN architectures that are being composed. However, for any set composition, there are many possible combinations of the members of the input sets. In order not to lose any potential products (as specified by the feature model), we need to keep all possible combinations, and so we have defined *family connectors* accordingly to perform these set compositions.

A *family connector* $F\text{-}Conn$ is defined as an n-ary function that takes a tuple of at least two X-MAN sets, and returns a product family, which is a tuple of an X-MAN set. The result of the composition performed by $F\text{-}Conn$ is a family of fully formed, executable products, each one in the form of an X-MAN architecture. The two $F\text{-}Conn$ connectors are $F\text{-}SEQ$ and $F\text{-}SEL$ corresponding to the X-MAN composition connectors *SEQ* and *SEL* respectively.

D. Family Filters

In order to handle *composition rules*, or *constraints*, that may be present in a feature model, we define a *family filter* as an operator on components composed by a family connector. A family filter removes products containing illegal combinations of components, from the family constructed by the family connector.

III. Constructing a Product Family

Clearly, by itself, FX-MAN just provides the building blocks for product families. However, the nature of these building blocks lends itself to the construction of product families whose architectures are feature-oriented in the sense that they are structurally isomorphic to the feature model.

At this stage, it should be obvious that the architecture of every product in FX-MAN (i.e. an X-MAN component, atomic or composite) is a tree, as composition is strictly hierarchical. This means that a product is hierarchically composed of components. Therefore if we use components to implement the features in the feature model, and construct a product family architecture in FX-MAN from these components, then the resulting architecture will be structurally isomorphic to the feature model. This is the basis of our approach to constructing product families in FX-MAN.

We construct a component c_i to implement each leaf feature f_i, and then hierarchically construct composite components C_i containing c_i, to implement parent features F_i of f_i. Variation operators can be applied at any level above the leaf level, and lead to permutations of composite components with features F_i and child features f_i. Finally, the tuples of X-MAN sets generated by variation operators are composed by family connectors into a family.

IV. Example

We have implemented a tool for our component model [14] and we have experimented with the construction of a family of Vehicle Control Systems (VCS) [20].

A VCS is a real-time, on-board system for controlling a motor vehicle. The key functionalities of VCS are captured in the feature model in Fig. 3.

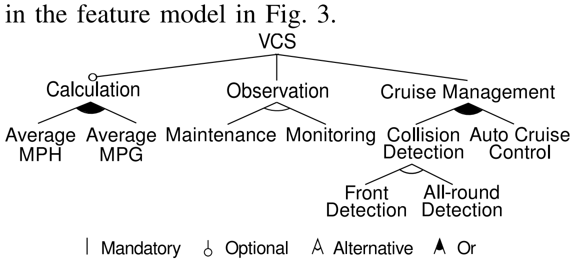

Figure 3: Feature model of VCS.

The feature model for VCS specifies that: (i) the Cruise Management feature is mandatory, which can provide Distance Detection or Auto Cruise Control, or both, and Distance Detection if present is either Front Detection or All-round Detection, but not both; (ii) the Observation feature is mandatory, which can yield either Maintenance or Monitoring, but not both; (iii) the Calculation feature is optional, which if present can provide Average MPH, Average MPG or both.

Following the VCS feature model, we now describe the steps needed to construct a family of VCS systems. The complete family is shown in Fig. **??**.

Step 1. The first step is to construct X-MAN components, atomic or composite, that implement the leaf (lowest level) features in the feature model; and then deposit them in the repository. There are seven leaf features, so we will construct seven X-MAN components: `AverageMPH`, `AverageMPG`, `Maintenance`, `Monitoring`, `FrontDetection`, `AllRoundDetection`, and `AutoCruise`.

Step 2. The second step is to apply variation operators defined in the feature model to the X-MAN components that have been constructed to implement the leaf features. To this end, we retrieve all the seven components from our repository, and apply the specified variation operators to them. The *Optional* operator applied to the tuple resulting from applying *Or* to `AverageMPH` and `AverageMPG` yields the tuple $F1$ = $\langle$\{`AverageMPH`\},\{`AverageMPG`\},\{`AverageMPH`\} $\oplus$ \{`AverageMPG`\}, $\emptyset$ $\rangle$. The *Alternative* operator applied to `Maintenance` and `Monitoring` gives the tuple $F2$ = $\langle$\{`Maintenance`\},\{`Monitoring`\}$\rangle$. The *Or* operator applied to the X-MAN set consisting of `AutoCruiseControl` and the tuple resulting from applying the *Alternative* operator to `FrontDetection` and `AllRoundDetection` yields the tuple of 5 X-MAN sets: $F3$ = $\langle$\{`AutoCruiseControl`$\oplus$`All-RoundDetection`\}, \{`AllRoundDetection`\}, \{`FrontDetection`$\oplus$`AutoCruiseControl`\}, \{`FrontDetection`\}, \{`AutoCruiseControl`\}$\rangle$.

Step 3. After generating variations, the last step is to compose the variations into a product family. It is worth noting that all the tuples of X-MAN sets specified by the variation points in the feature model have now been generated, but it remains to compose them into all the possible products specified by the feature model. Applying family connectors to these tuples of X-MAN sets will generate a product family, whose size depends on the cardinalities of these sets. The choice of family connectors is a design decision, however it will not affect the total number of products in the family. In this case the total number is 40. We choose to compose $F1$ and $F2$ into $F4$ with the family connector *F-Selector* because we want to allow the driver to choose any subset of the features: AverageMPH, AvergageMPG, Maintenance and Monitoring. Then we choose to compose $F4$ and $F3$ with *F-sequencer* to combine the driver's choice with the Cruise Management feature.

Step 4. Finally, the complete product family (Fig. 4) or a single member (e.g. Product 4 in Fig. 5) can be extracted.

V. Related Work

Our work in this paper is about a new component model that can be used to construct a product family from components (that represent products and product sub-families),

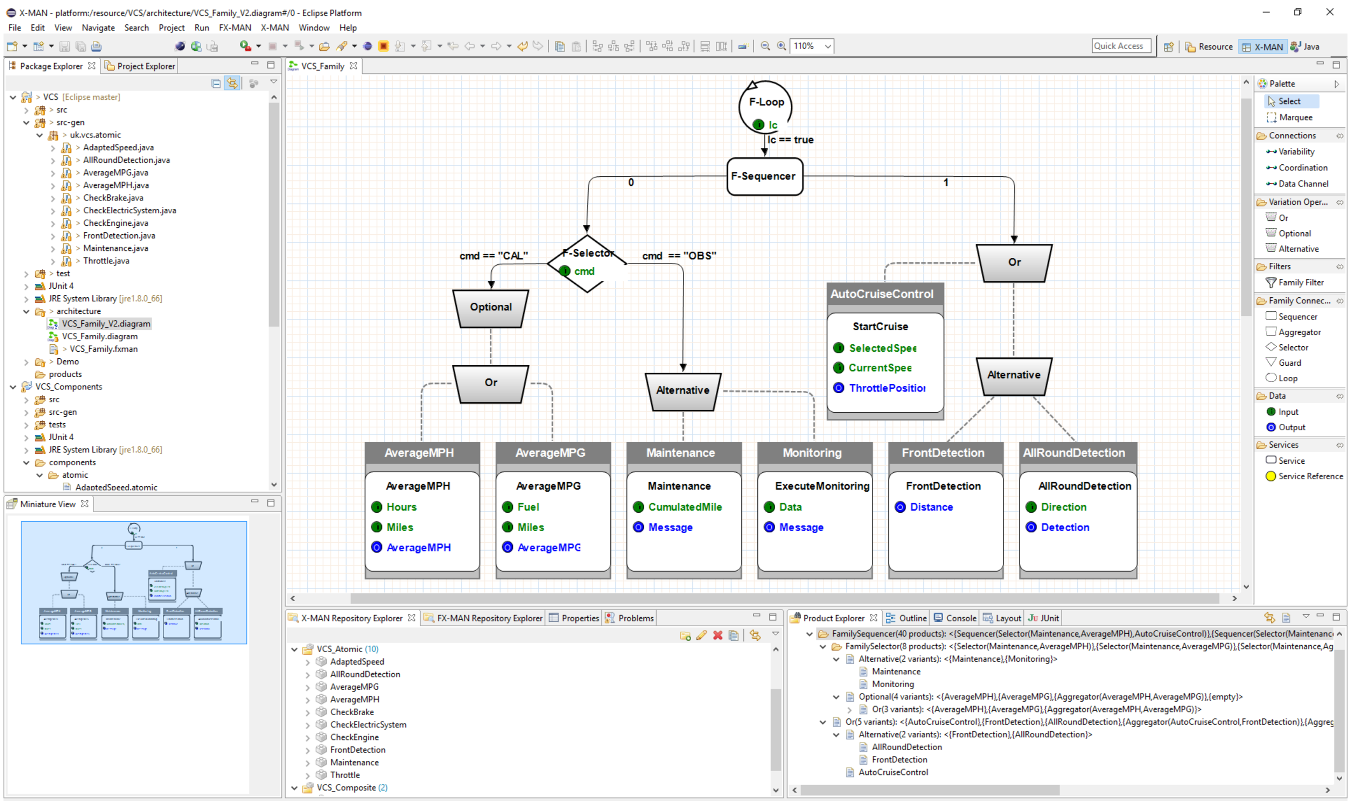

Figure 4: VCS product family.

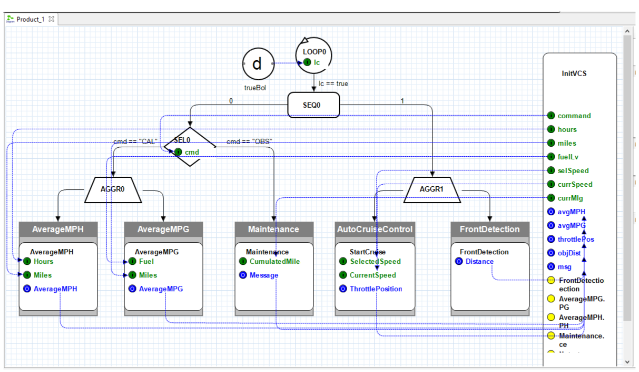

Figure 5: Product No.4.

variation operators (that represent variation points in a product family), and composition connectors that compose sub-families. An architecture created in our model contains a family of (sub-families of) fully formed, executable products.

This is in contrast to related work, which falls into two main categories: (i) component models (ii) variability handling approaches.

Figure 6 shows a comparison between FX-MAN and component models that define parametrised architectural templates. These models include: ADLARS [6], MontiArch^{HV} [18], ΔMontiArc[19], KobrA [5], Mae [30],Plastic Partial Components [28], xADL2 [13], Koala [35], Com [27], and Kumbang [4].

Component Model/ADL	Explicit Variation Points			Product Template/ Family
	Alt	Opt	Or	
ADLARS	×	×	×	Template
MontiArc^{HV} ΔMontiArc	×	×	×	Template
KobrA	×	×	×	Template
Mae	×	×	×	Template
Plastic Partial Components	×	×	×	Template
xADL2	×	×	×	Template
Koala	✓	×	×	Template
Com	✓	✓	×	Template
Kumbang	✓	✓	×	Template
Our model: FX-MAN	✓	✓	✓	Family

Figure 6: Component models and ADLs.

Some of these models do not define variation points explicitly, and express variability by other means. For example, MontiArch^{HV} [18] uses presence conditions, ΔMonti-Arc[19] use architectural deltas, while xADL2 [13] defines conditions in XML schemas. Other models do define some variation points explicitly. For example, Koala defines the *Alt* variation point explicitly (as a *switch* between components), but not *Opt* and *Or* (these can be simulated by parameters in the *diversity interface* of a component to change its internal structure). By contrast, FX-MAN explicitly defines the full standard set of variation points that appears in feature models: *Opt*, *Alt* and *Or*.

Having the full set of variation points explicitly enables

FX-MAN to be used to define architectures structurally isomorphic to the feature model in all cases. Conversely, the lack of the full set of explicit variation points means that the other component models can only define such architectures in a limited number of cases. Furthermore, an FX-MAN architecture allows to analyse a family, and its family members, at design time without the need of additional configuration. In other words, where other component model realise a template, FX-MAN realise a family architecture with explicit behaviour and variation points.

Variability Handling Approach	Metal Level?	Mandatory Features	Non-Mandatory Features	Configuration Points	Product Template/ Family
`Weaving' e.g. XWeave, Lee	✓	Code Base	Aspects	Pointcuts	Template
`Annotating' e.g. cpp, FArM	×	Code Base	Code Base	Annotations	Template
`Superimposition' e.g. Czarnecki, Apel	×	Artefact Fragment	Artefact Fragment	Presence Condition	Template
FX-MAN	×	Components	Components	Variation Points	Family

Figure 7: Variability management approaches.

In a wider context, SPLE methods and tools that do not construct architectures (or use a component model), rely on variability handling mechanisms. Figure 7 shows a comparison between FX-MAN and existing approaches to handle variability. There are three main categories of such approaches: (i) weaving-based (ii) annotation-based (iii) superimposition.

Weaving-based approaches [10], [15], e.g. XWeave [16] and AFM [1], manage variability by applying the principles of aspect-oriented programming [11] at the meta-level. Base models are varied by pointcuts and advices: the former define where to affect the base model, while the latter specify how to modify it. Product derivation is achieved by weaving the set of aspect models corresponding to a particular feature configuration.

Annotation-based approaches are widely used in industry [7] why they are very well supported through the commercial tools Gears [22] and pure::variants [9]. On the low level side the c-preprocessor (cpp), or FArM [33] are examples of such approaches. Here, artefacts as fragments of a code base are annotated with statements for example with *#ifdef*. Product derivation is achieved by removing fragments that do not reflect feature selection.

Superimposition [12], [3], [2] is the process of composing fragments of software artefacts (e.g. code, UML diagrams) by merging their corresponding substructures on the basis of nominal and structural similarity. Products are derived by merging only the fragments that satisfy their presence condition.

Like the component models in Figure 6, the key difference between all these variability handling approaches and FX-MAN is that they define a template for a product family, and not an architecture for a product family as in FX-MAN. Individual products have to be configured one at a time using the template.

VI. Discussion and Conclusion

The distinguishing characteristic of FX-MAN is its applicability to the construction of the architecture of a complete family of executable software products, together with the key advantage that the products can be analysed at design time without the need to be extracted. However, enumerating a complete product family is an NP-hard problem: for large-scale families with a high degree of variability, enumeration and extraction of a complete family is costly both in terms of computation time and memory. For practical purposes, a divide-and-conquer strategy might be necessary, to handle a large product family by decomposing it into sub-families. Happily this is possible in FX-MAN, due to its compositional nature, and its associated type system.

Another important aspect of compositionality is that FX-MAN can be used to compose families into bigger ones. This is possible because variation operators and family connectors can be applied at any level of composition on X-MAN sets (every product family is a tuple of an X-MAN set).

We are currently collaborating with pure::variants, the current market leader in variability management [7], in order to automate the mapping between *problem space* and *solution space*. This collaboration will enable us to evaluate our approach on larger real-world case studies, and we intend to do so.

Finally, our tool is available at http://www.click2go.umip.com/i/software/x_man.html.

References

[1] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. Trans. on *SE*, pages 162–180, IEEE, 2008.

[2] S. Apel, F. Janda, S. Trujillo, et al. Model superimposition in software product lines. In *Theory and Practice of Model Transformations*, pages 4–19, Springer, 2009.

[3] S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Soft. Comp.*, pages 20–35. Springer, 2008.

[4] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Adv. Eng. Inf.*, pages 23–40, Elsevier, 2007.

[5] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: the KobrA approach. In *SPL*, pages 289–309. Springer, 2000.

[6] R. Bashroush, T John Brown, I. Spence, et al. Adlars: An architecture description language for software product lines. In 29th Annual IEEE/NASA *SE Work.*, pages 163–173. IEEE, 2005.

[7] T. Berger, R. Rublack, D. Nair, et al. A survey of variability modeling in industrial practice. In 7th *Proc. of VaMoS*, page 7. ACM, 2013.

[8] T. Berger, D. Lettner, J. Rubin, et al. What is a feature?: a qualitative study of features in industrial software product lines. In 19th *Proc. of the SPLC*, pages 16–25. ACM, 2015.

[9] D. Beuche. Modeling and building software product lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 255–255. ACM, 2012.

[10] T. J. Brown, R. Gawley, R. Bashroush, et al. Weaving behavior into feature models for embedded system families. In 10th Proc. of SPLC pages 52–61, IEEE, 2006.

[11] S. Clarke and E. Baniassad. *Aspect-oriented analysis and design*. Addison-Wesley Professional, 2005.

[12] K. Czarnecki and M. Antkiewicz. *Mapping features to models: A template approach based on superimposed variants*. In 4th Proc. of GPCE,pages 422–437,Springer,2005.

[13] E. M Dashofy, A. van der Hoek, and R. N Taylor. A comprehensive approach for the development of modular software architecture description languages. In Trans. on *TOSEM*, pages 199–245, ACM 2005.

[14] S.Di Cola, K.-K. Lau, C. Tran, et al. An mde tool for defining software product families with explicit variation points. In 19th Proc. of *SPLC*, pages 355–360. ACM, 2015.

[15] P. Gilles, G. Vanwormhoudt, B. Morin, et al. Weaving Variability into Domain Metamodels. *Soft. & Sys. Modeling*, pages 361–383, July 2012.

[16] I. Groher and M Voelter. Xweave: models and aspects in concert. In 10th Proc. of *AOM*, pages 35–40. ACM, 2007.

[17] I. Groher and R. Weinreich. Integrating variability management and software architecture. In 10th Proc. of *WICSA/ECSA*, pages 262–266. IEEE, 2012.

[18] A. Haber, J.O. Ringert, and B. Rumpe. Montiarc – architectural modeling of interactive distributed and cyber-physical systems. Tech. Rep., RWTH Aachen, 2012.

[19] A. Haber, T. Kutz, H. Rendel, et al. Delta-oriented architectural variability using monticore. In 5th Proc. of *ECSA*, page 6. ACM, 2011.

[20] D. Hatley and I. Pirbhai. *Strategies for real-time system specification*. Addison-Wesley, 2013.

[21] N. He, D. Kroening, T. Wahl, et al. Component-based design and verification in X-MAN. In 11th Proc. of*ERTS*, 2012.

[22] C. Krueger and P. Clements. Systems and software product line engineering with biglever software gears. In 17th Proc. of *SPLC*, pages 136–140. ACM, 2013.

[23] K.-K. Lau. Software Component Models: Past, Present and Future. In 17th Proc. of *CBSE*, pages 185–186. ACM, 2014.

[24] K.-K. Lau and M. Ornaghi. Control encapsulation: A calculus for exogenous composition. In 12th Proc. of *CBSE*, pages 121–139. Springer-Verlag, 2009.

[25] K.-K. Lau and C. Tran. X-MAN: An MDE tool for component-based system development. In 38th Proc. *EUROMICRO SEEA*, pages 158–165. IEEE, 2012.

[26] N. Medvidovic and R. N Taylor. A classification and comparison framework for software architecture description languages. Trans. on*SE*, pages 70–93, IEEE, 2000.

[27] C. Park, S. Hong, K, -Ho Son, et al. A component model supporting decomposition and composition of consumer electronics software product lines. In 11th Proc. of *SPLC*, pages 181–192. IEEE, 2007.

[28] J. Pérez, J. Díaz, C. C. Soria, et al. Plastic partial components: A solution to support variability in architectural components. In Proc. of*WICSA/ECSA*, pages 221–230. IEEE, 2009.

[29] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[30] R. Roshandel, A. Hoek, M. Mikic-Rakic, et al. Mae a system model and environment for managing architectural evolution. Trans. on*TOSEM*, pages 240–276, ACM, 2004.

[31] I. Schaefer, R. Rabiser, D. Clarke, et al. Software diversity: state of the art and perspectives. Int. Jour. on *STTT*, pages 477–495, Springer, 2012.

[32] M. Sinnema, S. Deelstra, J. Nijhuis, et al. COVAMOF: A framework for modeling variability in software product families. In Proc. of *SPLC*, pages 197–213. Springer, 2004.

[33] P. Sochos, M. Riebisch, and I. Philippow. The feature-architecture mapping (farm) method for feature-oriented development of software product lines. In 13th Proc. of *ECBS*, pages 9–pp. IEEE, 2006.

[34] F. J van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action*. Springer, 2007.

[35] R. van Ommering, F. van der Linden, J. Kramer, et al. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, IEEE, 2000.