Feature-Oriented Development in Industrial Automation Software Ecosystems: Development Scenarios and Tool Support

Herbert Prähofer[1] Daniela Rabiser[2] Florian Angerer[2] Paul Grünbacher[2] Peter Feichtinger[2]
[1]Institute for System Software
Johannes Kepler University Linz, Austria
herbert.praehofer@jku.at

[2]Christian Doppler Laboratory MEVSS
Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria

***Abstract*—Due to increased market demands for highly customized and machine-specific solutions in manufacturing, industrial software systems are often developed as software product lines (SPL) and organized as software ecosystems (SECO) with internal and external developers composing individual solutions based on a common technological platform. In such settings, software development usually occurs in a multi-stage process: system variants initially derived from a platform are adapted and extended to meet specific requirements. This common approach, however, results in significant challenges for software development and maintenance. In this paper we review key challenges we have been observing when investigating our industrial partner's software ecosystems. We then present a feature-oriented development approach we have been developing to tackle those. Our approach is backed with static analysis methods to deal with system variants and versions created in software maintenance.**

I. Introduction

The engineering of industrial automation software is facing an increased demand for machine-specific solutions [1], [2] where automation programs are individually adapted and extended to optimize production. Therefore, industrial automation software systems are often developed as software product lines (SPL) [1], [3], [4] and organized as software ecosystems (SECO), a recent trend in software engineering [5], [6], [7], [8]. In SECOs individual software solutions are created by internal and external developers based on a common technological platform [6]. A prominent example is the Eclipse ecosystem: the platform core is developed by a globally distributed team of experts, while a cloud of external developers implements specific solutions based on the platform. In such a context, customer-specific solutions are often built in a multi-stage manner: system variants are first derived from the product line and then adapted and extended in a clone-and-own manner to meet the specific requirements of customers.

For instance, our industry partner Keba AG (www.keba.com) develops hardware and software platforms and solutions for industrial automation. The development of automation solutions is a multi-stage process involving different stakeholders: Keba develops and produces hardware and software platforms with associated tool support in various variants to meet the requirements for different market segments. The hardware and software platforms enable Keba's customers, usually OEMs of automation machines, to develop customized automation solutions for their products. Eventually, every machine is individually adapted and tuned for its particular application purpose, either by the OEM or directly by the end-user customer.

Software development in SECOs results in new challenges regarding software architectures, development processes [6], and tool support [9]. Specifically, it leads to many variants and versions, calling for effective variability management of components and customer solutions. Supporting this variability in the software, however, leads to increased design complexity and maintenance effort. Although some progress has been reported recently [2], [10], current languages and design approaches do not provide adequate support needed in industrial-size applications [11]. Moreover, the case study reported in [12] reveals that opening up development to an often unknown community of contributors leads to new problems for which no solutions exist so far.

Feature-oriented development [13], [14] has been proposed to address some of these challenges. However, although developers often think in terms of features, a feature-oriented view on the software is often hard to establish, as feature implementations usually span multiple subsystems and languages, and the connections are difficult to trace. This may result in developing the same or similar features when creating customer-specific solutions, often not exploiting the reuse potential. Furthermore, upgrading solutions to new platform releases becomes difficult if variants and versions in the ecosystems are not managed [12]. In a long-term research cooperation with our industry partner Keba AG, we are thus exploring a feature-oriented development approach [15] coupled with configuration-aware static program analysis techniques [16], [17]. Based on real world requirements provided by Keba AG, we are investigating techniques and implement prototypical tool solutions to improve software variability management, to increase the degree of reuse, and to provide support for software evolution and maintenance tasks.

In previous work, we have investigated key development and evolution challenges in industrial SECOs [11], [12], have described techniques and tools for feature-oriented modeling

978-1-5090-2870-2/16/$31.00 ©2016 IEEE

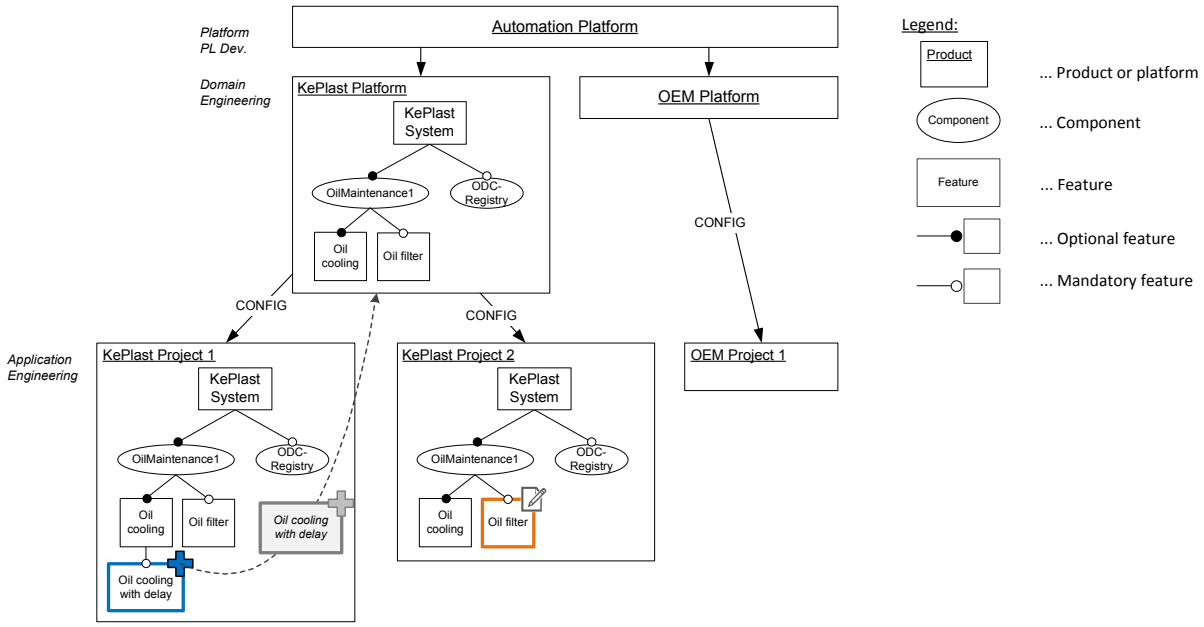


Fig. 1. Common engineering approach in SECOs.

of large-scale industrial applications [15], and methods for configuration-aware program analysis [16], [17]. In this paper, we now show how these different methods and tools work together in development and evolution scenarios of industrial SECOs. The paper is organized as follows: in Section II we discuss engineering challenges using an industrial case. Section III gives an overview of our feature-oriented development approach and the configuration-aware program analysis methods. In Section IV we discuss how the approach supports various development and evolution scenarios. The paper concludes with a summary and an outlook on future work.

II. Engineering Challenges in Industrial SECOs

A. Industrial SECO Development Process

In a recent exploratory case study, we investigated Keba's SECO to derive development characteristics and evolution challenges. Keba's development is organized as a multi-stage process relying on several platforms as shown in Fig. 1. The central pillar is the *automation platform* providing run-time systems and basic capabilities for automation solutions. It is organized as a product line allowing the derivation of different platform variants by selecting, composing, and configuring components. The platform variants then form the basis for diverse domain solutions developed by Keba or their customers, typically OEMs of automation machines.

An example of such a domain solution is KePlast [18], which provides a comprehensive solution for the automation of injection molding machines. It consists of a configurable control software framework implemented in a proprietary dialect of the IEC 61131-3 standard, a visualization system written in Java, respective programming tools, and a configuration tool supporting the interactive configuration of solutions based on existing components and variants. The platform exists in multiple variants for different market segments. Based on the generic domain solution, concrete applications are developed by application engineers to meet specific customer requirements: they first derive a basic solution from the domain platform using the configuration tool, and then adapt and extend the software. For instance, engineers may add new features or modify existing ones, thereby often evolving the KePlast platform. Moreover, our exploratory case study [11] and other work [19] shows that OEMs have adopted a similar multi-stage development approach.

B. Industrial SECO Development and Evolution Challenges

We derived development and evolution challenges in workshops with managers and project leaders, interviews with developers, and archival analyses [11], [12]:

Lack of feature-oriented views on software. Features are an important concept within Keba's SECO, used to support the communication of developers, service engineers, product managers, and sales personnel. However, a feature-oriented view on the software, showing which features are implemented in which parts of the system, is currently not available. This makes it difficult for developers to locate the relevant artifacts when making changes. Moreover, role-specific perspectives regarding the features of a platform usually differ considerably, which can lead to misunderstandings, e.g., when checking the actual implementation of a feature defined in a product map.

Further, feature implementations usually are cross-cutting and span multiple subsystems, often implemented in different programming languages. For example, it is very common that a feature handling a specific hardware equipment comprises a control part implemented in the IEC 61131-3 language a visualization implemented in Java. Developers have

difficulties comprehending these different aspects of a feature as no suitable cross-cutting representations exist.

Lack of systematic reuse of feature implementations: As customer requirements are highly specific, customer-specific products are created by adapting and extending the solution in a clone-and-own reuse approach. These products contain potentially reusable features which might be helpful in future products. However, developers are hardly aware about commonalities and variability of existing products and deviations from the platform are hardly documented. This makes it difficult to see which features are implemented in a product, or how features could be reused in other products.

Lack of support for platform evolution. When evolving the platform, e.g., when implementing a new feature or fixing a bug, developers have difficulties in predicting the impact of their changes on other SECO platforms and product variants. It is often hard to determine for developers if a change is valid for all, or even only for the most important system variants. Besides, Keba's developers have to consider multiple subsystems and programming languages when analyzing the impact of a change.

Likewise, merging existing applications and product variants with new platform releases often means a significant effort. Application engineers building customer-specific applications first derive a basic application from a platform and then make arbitrary changes and additions to meet a customer's requirements. Upgrading these specific product variants after a new platform release is highly challenging, as the new and updated platform features have to be merged with the changes made in the variants.

III. Feature-oriented Approach and Tool Environment

We have been developing a feature-oriented modeling environment [15], backed with configuration-aware static analysis methods [16], [17], to support SECO development in an industrial context. Specifically, we use a multi-purpose, multi-level feature modeling approach augmented with feature-to-code mappings for relating feature models with source code, represented as a code model.

Fig. 2 show the basic concepts and how they are related. Models can be organized in several different spaces based on their purpose: e.g., there can be separate models for product management, system configuration, or system development. Models are structured in components, which are arranged hierarchically, i.e., components can have sub-components. Components adopt the idea of configurable units proposed in the Common Variability Language (CVL) [20], i.e., they provide variability specifications in form of cardinality-based feature models [21] defining the developers' understanding of the components' capabilities and variability.

The code model in the form of an abstract syntax tree (AST) provides a full representation of the system implementation, also including such elements a configuration settings or task definitions. The AST representation is technology-independent, i.e., it can represent systems implemented in different programming languages, thus supporting analyses

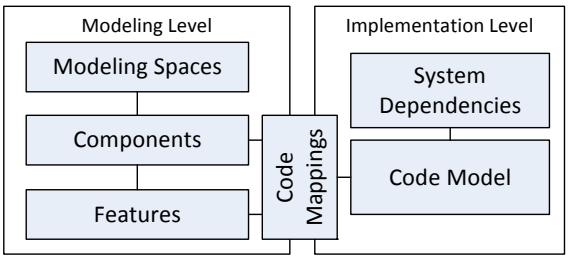


Fig. 2. Modeling concepts and program analysis approach.

across multiple languages. Moreover, a system dependence graph (SDG) is computed from the AST representation, which encodes all control flow and data flow dependencies in a program (cf. [22]) to allow conducting global dependency analyses in a program. Further, the system dependence graph is augmented with variability information to make the analysis configuration-aware [16]. Specifically, the control and data flow edges in the SDG are augmented with presence conditions encoding which control or data flows are valid for particular configurations.

The connections between feature models and code model are defined via code mappings representing how components and features are implemented in the program. That means, components provide links to related implementation artifacts, e.g., software modules. Analogously, features are mapped to implementation artifacts representing the implementation of features, usually taking the form of a set of AST nodes. As our AST representation is technology-independent, feature implementations can span multiple modules, subsystems and programming languages.

Our modeling and analysis approach is implemented in a tool environment based on the Eclipse RCP platform. The modeling environment has been implemented as an extension of the feature-oriented development environment FeatureIDE [23]. In particular, we have introduced multiple modeling spaces and hierarchical components. The AST representation has been implemented based on the Abstract Syntax Tree Meta-Model (ASTM) standard [24] from the Object Management Group and its Modisco implementation [25]. ASTM provides a language-independent foundation for building AST models of different languages. We have extended ASTM for representing the IEC 61131-3 programs of our industry partner and used Modisco's AST for representing Java programs. In this way, our ASTM provides an integrated representation of IEC 61131-3 and Java programs in one model. Based on the ASTM representation, we used the Soot analysis framework [26] for computing a global SDG. For that purpose, the IEC 616131-3 programs are translated to Jimple code, which is the input format for Soot [27].

IV. Development Scenarios and Tool Support

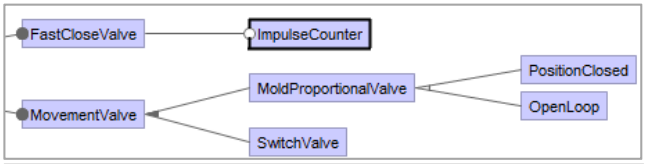
Our modeling and analysis approach supports different development and evolution tasks. In this section we describe selected scenarios illustrating our tool support.

A. *Semi-automatic computation of feature-to-code mappings*

As outlined above, feature-to-code mappings build the links between features and their implementation in the form of sets of program elements represented as AST nodes. The mappings are established by the developer creating the feature model. We support creating these mappings semi-automatically by static analysis methods.

In particular, for optional and alternative features, variability implementation mechanisms must be provided in the code base that allow activating the respective code when selecting the feature. For example, in the case study system IS_LINKED is an intensively used variability mechanism: program variables usually represent the endpoints to hardware equipment, e.g., if a variable is used to provide the values of a sensor. For optional hardware equipment, the variable declaration is used as variation point in the software, i.e., when the hardware option is included in the machine, a corresponding variable declaration is included in the program. The actual implementation of a feature, however, is not only the variable declaration but all program parts that become active by the declaration. Specifically, the program will contain conditional statements testing the presence of the variable declaration (using the built-in function IS_LINKED) and only conditionally execute the code responsible for handling that optional equipment (see example code in Fig. 3). Thus, we use the SDG to compute the program parts enabled by the variable declaration and propose it as the feature's implementation to the modeler. Fig. 3 gives an example: the feature ImpulseCounter is optional and the variable di_ImpulseInput represents a variable to a sensor. Only, if it is available (function IS_LINKED), the highlighted code becomes active. Thus, it is considered to be part of the implementation feature ImpulseCounter.

The automatically computed feature implementation can, as our case study shows, comprise multiple locations and span several system parts, often implemented in different languages. We provide similar feature detection mechanisms for other types of run-time variability mechanisms, e.g., configuration settings.




```
POSTUPDATE ALGORITHM pMeasureManager ON TaskAnalog WITH sv b

IF IS_LINKED(di_ImpulseInput) THEN
    IF NOT sv_bDeviceReady THEN
        // Reset the stand still detection and the impulse co
        // And Start the measuring algo paMeasure
         fbTStandStill(IN := FALSE);
         sv_iThisStepsActImpulses := 0;
         START_PROCESS_ALGORITHM(paMeasureImpulses);
    ELSE
        STOP_PROCESS_ALGORITHM(paMeasureImpulses);
    END_IF;
```

Fig. 3. IS_LINKED code mapping: Excerpt from feature model and code mapping for the selected feature ImpulseCounter.

B. *Marking inactive code in application development*

In the multi-stage development process of industrial SECOs, developers create solutions by first selecting required components, then setting configuration options, and finally building the customer-specific solution by changing features and implementing new features at source code level. However, when using run-time variability mechanisms such as IS_LINKED or load-time configuration parameters, it is not immediately obvious which program parts are active in a specific configuration. This means that developers have to read and understand potentially a lot of inactive code during maintenance. Our case study evaluations based on the KePlast product line showed that inactive code represents a significant portion of the overall code of a product variant [16]. Further, we showed that for many options, the inactive code is scattered over many files.

We thus presented an approach for automatically identifying (and hiding) inactive code given a concrete product configuration [16]. The inactive code detection approach uses the configuration-aware SDG which globally considers all control and data dependencies in a program. For example, when detecting that a procedure is inactive, i.e., not called in a current product variant, it will also mark the procedures called by this procedure as inactive (evidently only if they are not called from other active call sites). The inactive code can then be greyed out in the tool environment, thereby allowing the developers to better focus on the source code relevant in a specific product variant (see Fig. 4).

C. *Configuration-aware change impact analysis*

When evolving a platform, the domain engineers face the difficulty to determine the impact of their changes on all possible variants of the product family. Classical change impact analysis (CIA) techniques [28] allow identifying the possible consequences of a change or estimating what needs to be modified to accomplish a change. However, they fail when it comes to highly variable and configurable systems, as the generation and consideration of all possible variants is infeasible, even for small systems.

Thus, in [17] we developed a configuration-aware CIA method, which uses the variability information encoded in the conditional SDG and propagates it along the. The result of this propagation are annotations for program elements in the form

```
IF IS_LINKED(di_ImpulseInput) THEN
    // Start and stop the paMeasure algo depending on
    // sv_bDeviceReady flag (postupdate var for this algo),
    // which is false everytime the movement is active
    IF NOT sv_bDeviceReady THEN
        // Reset the stand still detection and the impulse co
        // And Start the measuring algo paMeasure
         fbTStandStill(IN := FALSE);
         sv_iThisStepsActImpulses := 0;
         START_PROCESS_ALGORITHM(paMeasureImpulses);
    ELSE
        STOP_PROCESS_ALGORITHM(paMeasureImpulses);
    END_IF;
END_IF;
```

Fig. 4. Inactive code pruning: code not called in the current product configuration is shown in gray.

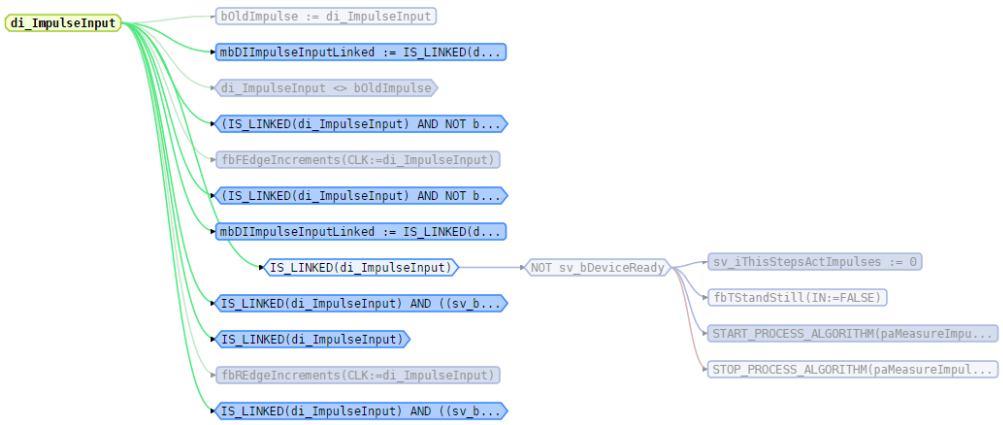


Fig. 5. SDG Browser showing change impact (inactive branches grayed out).

of logical combinations of configuration options, which encode for which configuration the statement will be executed.

The configuration-aware CIA approach can be useful when evolving the platform as well as when developing specific products. It allows a domain developer evolving the platform to determine which configurations are affected by a change impact. Further, it can support an application engineer working on a concrete product, as the change impact will be smaller when considering a specific product configuration. Fig. 5 shows a change impact in a concrete product configuration using our interactive SDG Browser tool. The graph shows the impacted program elements for a selected variable. The change impact considers the concrete product configuration. Inactive statements are grayed out. In [17] we showed that our configuration-aware CIA method is effective in supporting both domain engineers and application engineers. We have also shown that the complexity of the computed variability conditions requires tool support. Further, we have shown that the size of the change impact can be reduced significantly when considering concrete product configurations.

D. *Selective change impact analysis for feature updates*

An important maintenance and evolution scenario outlined in Section III is updating an existing product with new versions of features from the platform. Although a feature may have been tested in the platform, the specific changes made in the product may be in conflict with the new version of the feature. The question is: can the product be updated with the new feature implementation or how must the product be adapted to be compatible with the new feature implementation?

The assumption is that only the changed program parts of the product have to be considered, as the other parts are identical to the platform code. Thus, the developer has to check if there is an impact from the new version of the feature to the changed parts of the product. This can be accomplished by a selective CIA. Moreover, when considering the current product configuration, the configuration-aware CIA possibly allows to exclude impacts which are not relevant.

E. *Configuration-aware control and data flow analysis*

As outlined above, the SDG encodes all the control flow and data flow dependencies in a program. Hence it also can reproduce all potentially possible execution scenarios in a program run which allows answering questions important for maintenance. For example, one may ask how a particular variable can get assigned or why a particular variable has a specific value.

We use the interactive SDG Browser tool to support such kind of analyses (cf. Fig. 6): a developer can select a particular variable and then decide to view all assignment operations to this variable, also considering assignments based on references and pointers. The SDG Browser tool allows to selectively follow the assignments statements back to the preceding statements and conditional branches, to procedures and procedure calls to finally find the call of the main routine.

In this way, a developer is able to see all the possible executions leading to an assignment of this variable. Again, considering a concrete product configuration, the configuration-aware SDG allows to prune away and gray out all the executions not feasible in the current configuration.

V. Conclusion

Industrial software systems are frequently organized as software ecosystems (SECOs) where internal and external developers jointly build customer-specific solutions in a multi-stage process, based on a common technological platform. In this paper we have discussed an approach for supporting SECO engineering in the industrial context. The approach adopts a

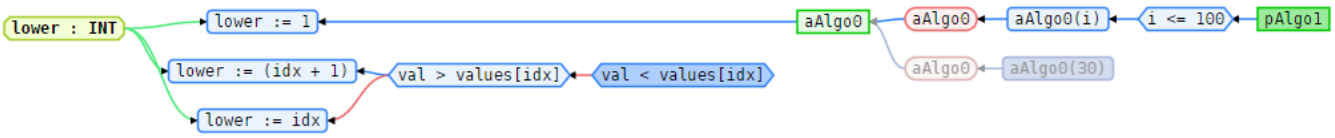


Fig. 6. SDG browser showing variable assignment (inactive branches grayed out).

feature-oriented modeling approach, which is backed by static program analysis techniques considering the variability of the platform and the configuration of products. The approach has been tailored based on a detailed analysis of development and evolution challenges of our partner company Keba.

Then the paper illustrated various scenarios which show how the approach supports development and evolution in SECO engineering. We have illustrated a technique to semi-automatically computing feature-to-code mappings, a method for pruning inactive code in product configurations, a change impact analysis method which uses the variability of the software platform, an application of the change impact analysis method for selective feature updates, and a method for answering why questions in a variable software system. However, the scenarios discussed in this paper only comprises a representative selection and many more exist. For example, [29] presents an approach for improving the developer awareness of feature implementation and thus increasing reuse. Another method we are currently exploring is the extraction of a feature implementation with all its dependencies from a product solution to facilitate its integration in an another product.

ACKNOWLEDGMENT

This work has been conducted in cooperation with Keba AG, Austria, and was supported by the Christian Doppler Forschungsgesellschaft, Austria.

REFERENCES

[1] B. Vogel-Heuser, J. Fuchs, S. Feldmann, and C. Legat. "Interdisciplinary product line approach to increase reuse." *at-Automatisierungstechnik*, vol. 63, no. 2, pp. 99-110, 2015 (in German).

[2] C. Legat, J. Folmer and B. Vogel-Heuser. "Evolution in industrial plant automation: A case study." *Proc. 39th Annual Conf. of the IEEE Industrial Electronics Society*, pp. 4386-4391, 2013.

[3] P. Clements and L. Northrop. *Software product lines: practices and patterns*. SEI Series in Software Engineering, 2002.

[4] F. Damiani, I. Schaefer, and T. Winkelmann, „Delta-Oriented Multi Software Product Lines," *Proc. of the 18th Int'l Software Product Line Conference* (SPLC 2014), pp. 232–236, 2014.

[5] J. Bosch,. "Software ecosystems: Taking software development beyond the boundaries of the organization.", *Journal of Systems and Software* vol. 85, no. 7, pp. 1453-1454, 2012.

[6] J. Bosch, and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems." *Journal of Systems and Software*, vol. 83, no. 1, pp. 67-76, 2010.

[7] K-B. Schultis, C. Elsner, and D. Lohmann. "Moving Towards Industrial Software Ecosystems: Are Our Software Architectures Fit for the Future?" *4th PLEASE Workshop at ICSE*, pp. 9-12, 2013.

[8] R. P. dos Santos, and C. M. Lima Werner, "A Proposal for Software Ecosystems Engineering," *3rd Int'l Workshop on Software Ecosystems* (IWSECO-2011), pp. 40-51. 2011.

[9] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A sense of community: A research agenda for software ecosystems". *Companion Int. Conf. on Software Engineering* (ICSE 2009 Companion), pp. 187-190. 2009.

[10] M. Kowal, C. Legat, D. Lorefice, C. Prehofer, I. Schaefer and B. Vogel-Heuser, "Delta modeling for variant-rich and evolving manufacturing systems," *Proc. 1st Int'l WS on Modern Software Engineering Methods for Industrial Automation* (MoSEMInA 2014), Hyderabad, India, pp. 32-41, 2014.

[11] D. Lettner, F. Angerer, H. Prähofer and P. Grünbacher. "A case study on software ecosystem characteristics in industrial automation software." *Proc. of the 2014 International Conference on Software and System Process*, pp. 40-49, 2014.

[12] D. Lettner, F. Angerer, P. Grunbacher and H. Prahofer. "Software evolution in an industrial automation ecosystem: An exploratory study." *Proc. 40th EUROMICRO Conf. on Software Engineering and Advanced Applications* (SEAA), pp. 336-343, 2014.

[13] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration using feature models." Proc. *Int'l Software Product Lines*, pp. 266–283, 2004.

[14] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: A comparison of variability modeling approaches," *Proc. 6th Int'l Workshop on Variability Modelling of Software-Intensive Systems*, Leipzig, Germany, pp. 173–182, 2012.

[15] D. Lettner, K. Eder, P. Grünbacher, and H. Prähofer. "Feature Modeling of Two Large-scale Industrial Software Systems: Experiences and Lessons Learned." *Proc. ACM/IEEE 18th Int'l Conf. on Model Driven Engineering Languages and Systems*, Ottawa, pp. 386-395, 2015.

[16] F. Angerer, H. Prähofer, D. Lettner, A. Grimmer, and P. Grünbacher. "Identifying inactive code in product lines with configuration-aware system dependence graphs." *Proc. of the 18th Int'l Software Product Line Conference* (SPLC 2014), pp. 52-61, 2014.

[17] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher, "Configuration-Aware Change Impact Analysis." *30th IEEE/ACM Int'l Conf. on Automated Software Engineering* (ASE 2015), Lincoln, Nebraska, USA, pp. 9–13, 2015.

[18] D. Lettner, M. Petruzelka, R. Rabiser, F. Angerer, H. Prähofer, and P. Grünbacher. "Custom-developed vs. model-based configuration tools: Experiences from an industrial automation ecosystem." *17th Int'l Software Product Line Conference (SPLC '13) co-located workshops*, pp. 52-58, 2013.

[19] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, "Opportunities and challenges of static code analysis of IEC 61131-3 programs." *Proc. IEEE Int. Conf. on Emerging Technologies and Factory Automation* (ETFA'12), pp. 1-8, IEEE, 2012.

[20] CVL: Common variability language (2012). http://www.omgwiki.org/variability/doku.php?id=start&rev=1351084099, [accessed 27-04-2015].

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

[22] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs." *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol. 12, no. 1, pp. 26–60, 1990.

[23] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development." *Sci. Comput. Program*., vol. 79, pp. 70–85, 2014.

[24] OMG, *Abstract syntax tree metamodel* (*ASTM*), TCS, IBM and others, Tech. Rep., 2007.

[25] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[26] P. Lam, E. Bodden, O. Lhoták and L. Hendren, "The Soot framework for Java program analysis: a retrospective," *Cetus Users and Compiler Infastructure Workshop* (CETUS 2011), 2011.

[27] A. Grimmer, F. Angerer, H. Prähofer, P. Grünbacher, "Supporting program analysis for non-mainstream languages: Experiences and lessons learned," Proc. 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), Osaka, Japan, 2016.

[28] R. S. Arnold, *Software Change Impact Analysis*, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[29] D. Lettner and P. Grünbacher, "Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution", *Proc. 9th International Workshop on Variability Modelling of Software-intensive Systems* (VaMoS 2015), pp. 11-18, 2015.