2014 IEEE International Conference on Software Maintenance and Evolution

Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems

Lukas Linsbauer*, Florian Angerer†, Paul Grünbacher*†, Daniela Lettner†,
Herbert Prähofer‡, Roberto E. Lopez-Herrejon* and Alexander Egyed*
*Institute for Software Systems Engineering, †Christian Doppler Laboratory MEVSS, ‡Institute for System Software
Johannes Kepler University Linz, Austria
{lukas.linsbauer, florian.angerer, paul.gruenbacher, daniela.lettner, herbert.praehofer, roberto.lopez, alexander.egyed}@jku.at

***Abstract*—Software engineering methods for analyzing and managing variable software systems rely on accurate feature-to-code mappings to relate high-level variability abstractions, such as features or decisions, to locations in the code where variability occurs. Due to the continuous and long-term evolution of many systems such mappings need to be extracted and updated automatically. However, current approaches have limitations regarding the analysis of highly-configurable systems that rely on different variability mechanisms. We present a novel approach that exploits the synergies between program analysis and diffing techniques to reveal feature-to-code mappings for highly-configurable systems. We demonstrate the feasibility of our approach with a set of products from a real-world product line in the domain of industrial automation.**

I. Introduction

Variability plays an essential role in most software systems today which need to support a wide range of different customer requirements. Research on variable software systems has progressed significantly, e.g., in the fields of software product lines or feature-oriented software development. The existing techniques frequently rely on feature-to-code mappings to relate high-level variability abstractions such as features or decisions to *variation points* – the locations in artifacts where variability occurs.

However, software product lines are rarely planned and developed from scratch. Instead they are typically the result of maintaining and evolving code bases over many years. This means that feature-to-code mappings either do not exist or are frequently outdated. Researchers have thus developed different automated approaches for recovering these mappings and traces. For instance, Xue et al. [1] present an approach to improve feature location in product variants by exploiting commonalities and differences of product variants. Rubin et al. [2] suggest heuristics for improving the accuracy of feature location techniques by analyzing multiple product variants.

However, these approaches provide only partial answers for the maintenance and evolution of real-world systems: (i) real-world product lines still face clone-and-own reuse, i.e., the products are not derived in a systematic way from the product line but cloned and extended to provide the required functionality. This means that existing feature-to-code mappings become inaccurate making automated analyses challenging. (ii) in variable software systems different variability mechanisms are used in combination. For instance, systems frequently rely on static annotation-based mechanisms such as preprocessor techniques as well as runtime variability approaches allowing the adaptation of systems during operation. Variability-aware program analysis can help to compute feature-to-code mappings in such contexts. However, current analysis approaches often make assumptions about form of variability, i.e., they often assume annotation-based PLs with preprocessors [3]. Program slicing is another way to identify feature-to-code mappings. For instance, newer program slicing approaches such as Hammer et al. [4] handle runtime variability by attaching path conditions to System Dependence Graphs (SDGs) to improve the precision of slices. However, these approaches do not scale to large programs as path conditions need to be extracted for nearly every conditional statement.

We recently conducted an empirical study on the evolution of product lines (PLs) in an industrial ecosystem confirming challenges (i) and (ii) [5], [6]. Our goal is thus to improve clone-and-own reuse by automatically recovering variability traces in mixed-variability systems. Our research is based on our earlier work: the *Extraction and Composition for Clone-and-Own (ECCO)* approach [7], [8] allows recording variability traces in clone-and-own product lines—cf. challenge (i). The *Configuration-Aware Program Analysis (CAPA)* approach [9] supports mixed variability PLs via a conditional system dependence graph with presence conditions to represent different types of variability in a system—cf. challenge (ii). In this paper we report our progress in integrating these two strands of research.

More specifically, we use CAPA to statically analyze product variants to create conditional SDGs. ECCO then uses a diffing algorithm to map features to the code base. The feature traces are identified based on the differences between the product variants. The results of this analysis can be used to integrate different product clones into a single system representing the variability mined in the product variants. We demonstrate the feasibility of our approach by applying it to a SPL in the domain of industrial automation. The results of our preliminary evaluation show that our approach improves the recovery of feature-to-code mapping in mixed variability, clone-and-own product lines.

II. Background

Variability in software can be implemented by means of different approaches. Static variability mechanisms determine at compile time what code to include in a program variant. An

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.67

example are make files that only compile and link files required for a specific variant. Another example are preprocessors determining what lines of source code to include. Runtime variability mechanisms allow to define the code to be executed in a program variant during execution by adapting its features. An example are configuration files that are processed during program execution. Variability mechanisms are frequently combined, e.g., when using coarse-grained static variability mechanisms to select the modules needed for the required features together with a runtime variability mechanism to determine fine-grained variability of program behavior.

We integrate program analysis and diffing techniques to improve the automated recovery of feature-to-code mappings: we first use CAPA to analyze mixed-variability program variants with runtime variability. We then use ECCO to extract feature-to-code traces and trace dependencies based on this input.

A. Configuration-aware Program Analysis (CAPA)

Runtime variability is often achieved by configuration options tested in conditional statements to enable or disable features. This means that the source code of one product variant may contain the implementation of all features, even if certain features are not part of the product variant. The CAPA approach [9] enables the identification of source code that is relevant for a specific product variant. Specifically, the approach analyzes the conditional statements in a program that test configuration options read from configuration files as initial *seeds* or traces for the implementation of features. CAPA analyzes the source code of a single product variant and builds a system dependence graph (SDG). Then, it uses architectural knowledge to identify conditional statements representing variation points in the system. The variability mechanisms used in the system are formulated as AST patterns to allow finding the corresponding code locations.

For example, in Java it is very common to use class `java.util.Properties` to access configuration options stored in property files. These files represent product configurations as they define values of configuration options. The values of configuration options are typically accessed by specific methods. For instance, if domain experts know that the `getProperty` method is used for that purpose it is possible to define an AST pattern matching such calls. Domain experts must further define how to map such patterns to features in an existing feature model. For instance, they may specify a generic transformation of the configuration option's name to a feature name, `featureName("config_feature") = "feature"`. The corresponding feature condition of the variation point is then attached to the corresponding edges in the SDG resulting in a *conditional* SDG (CSDG). Finally, CAPA performs a reachability analysis on the CSDG using the actual product configuration resolving the variation points and marks unreachable nodes as inactive. This is similar to the idea of Conditional Program Slicing, which involves the use of symbolic execution to compute slices with respect to specific input variable values [10].

Listing 1 shows an example using Java's `Properties` class to implement variability in the software system. The method *doSomething* performs some operation and additionally contains code for logging. However, the logging code is optional and can be activated by setting the configuration option *config_logging* to `enabled`. Once a domain expert specified this pattern, CAPA is able to identify active code for the current product configuration (in the example encoded in the properties file *configuration.properties*).

```
public class A {
  private Properties prop =
              Properties.load("configuration.properties");

  public void doSomething() {
    // method implementation

    if("enabled".equals(
          prop.getProperty("config_logging"))) {
      log(LogLevel.INFO, "some log message");
    }
  }

  public void log(LogLevel l, String msg) {
    // ...
  }
}
```

Listing 1. Sample code using Java's Properties class as variability mechanism.

B. Extraction and Composition for Clone-and-Own (ECCO)

ECCO compares different program variants to extract feature-to-code traces, interactions between features, and dependencies between traces. ECCO assumes as input a set of n product variants about which two things are known: (i) the source code that implements each product variant and (ii) the set of features that each program variant provides. It is not important how these product variants came to be. They can be separately maintained variants created by a manual clone-and-own approach or product variants generated using a more structured approach like preprocessor annotated source code.

As output ECCO provides traces defining how source code is related to features. For instance, a trace defines for a particular part of the code which features it implements likely, at most, or certainly not [7]. ECCO also considers *feature interactions* (i.e. traces that refer to source code that is present only when all interacting features are present), and *negative features* (i.e., traces that refer to source code that is included in case of the absence of a feature). Furthermore, ECCO also extracts dependencies *between* traces. Traces (e.g., to a feature) can require other traces (e.g., to another feature) to function. For example, a trace A that contains a statement calling a method which is part of another trace B depends on that trace B. Based on this information ECCO constructs a simple variability model that can be used as a starting point for manually defining a feature model.

Lastly, ECCO also comes with a composition tool that can use the previously extracted information to compose product variants with given sets of features they shall implement.

III. MVA Approach

Figure 1 depicts the high-level view of our new *Mixed-Variability Analysis (MVA)* approach, which combines CAPA and ECCO to recover feature-to-code mappings in mixed-variability systems. In mixed-variability systems with static and runtime variability mechanisms, the product variants may also contain feature code not part of a product variant. Hence, MVA first uses CAPA in a preprocessing step to remove this dead code of the n product variants and generates n

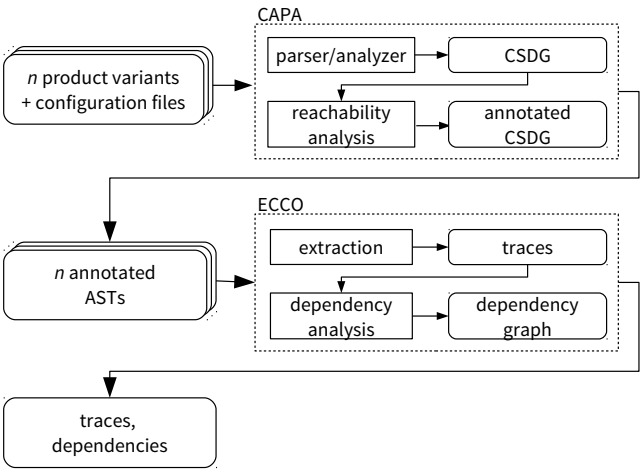

Fig. 1. Recovering feature-to-code mappings with mixed-variability analysis and diffing of product variants.

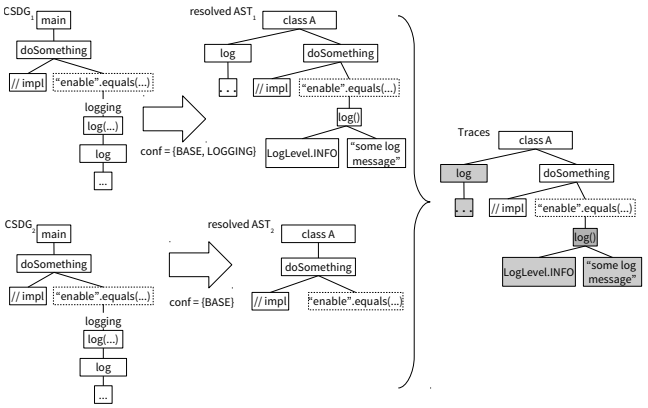

Fig. 2. CAPA computes CSDGs for each product variant. It performs a reachability analysis on the CSDGs using the corresponding product configurations to identify inactive code in product variants. If a CSDG node is unreachable, the corresponding AST node is marked inactive (left side). The traces generated by ECCO are shown on the right side.

annotated ASTs which are then further processed by ECCO. First, CAPA parses the source code of every product variant and creates ASTs. Next, CAPA computes intra-procedural program dependence graphs (PDGs) based on intra-procedural control and data flow information. The PDGs represent the same information as the SDG but are limited to individual procedures. The approach uses Harrold et al.'s algorithm [11] computing PDGs directly based on the AST. The SDG is then created by linking individual PDGs.

CAPA then identifies variation points by matching dedicated patterns in the AST and generating conditions from these variation points. The conditions are attached to the corresponding edges in the SDG resulting in the conditional SDG (CSDG), which represents the basis for identifying the code which can be executed and code which cannot be executed in a given product variant.

Finally, CAPA identifies unreachable nodes in the CSDG by performing a reachability analysis for a specific product variant. First, all nodes in CSDG are marked as inactive. The reachability analysis algorithm starts at the root procedure node. It traverses the CSDG by following the edges and marks all reached nodes as active. Every time an edge with a presence condition is visited, the edge is followed only if the condition is satisfied regarding the values of the variables representing the product configuration. In order to do so, the concrete product configuration is transformed into assignments to the variables used in the presence conditions. Therefore, this transformation is a complementary part of the step of extracting the presence conditions and establishes a semantic link between a configuration and the statements in the program.

Until now, the information about dead code in a concrete product configuration is available in the annotated CSDG. ECCO expects as input a tree structure representing the product variant's source code. Therefore, the information concerning the active and inactive CSDG nodes is transferred to the corresponding statement nodes in the AST. Finally, CAPA analyzes the usage dependencies of every type using the type information of the AST. If a type is not used by an active AST node, the type's declaration will also be marked inactive. ECCO then uses the annotated ASTs as input and computes the traces and the dependencies between the traces.

Figure 2 illustrates the ASTs resulting from two different product variants that statically have the same source code. Only the first product variants contains the feature *LOGGING* which can be enabled with a configuration. However, both product variants statically have the same source code which would result in the the same ASTs and would make it infeasible for ECCO to create a useful feature-to-code mapping. CAPA thus applies the concrete product configuration, stored in text-based configuration files, and performs the described reachability analysis on the CSDG. In the first product variant, all CSDG nodes are reachable. In the second product variant, the CSDG nodes representing the implementation of feature *LOGGING* are not reachable since the presence condition `logging` evaluates to false. CAPA therefore generates two different ASTs for the two product variants and ECCO can compute useful feature-to-code traces.

IV. Preliminary Evaluation

In this preliminary evaluation we show first indications that the extracted traces and the dependencies among them are correct and useful in real-world scenarios. For this purpose we discuss first results of applying our approach to a real-world industry case study.

A. Case Study

Our industry partner Keba AG (http://www.keba.com) is developing and producing hardware, software, and tools for industrial automation solutions including injection molding machines, robotics, and heating system control. Keba develops and maintains several heterogeneous platforms that exist in numerous variants to meet customer requirements in different market segments. Keba's engineers follow a flexible development strategy combining staged configuration [12] and ad-hoc clone-and-own reuse [13] for creating customized system solutions based on the KePlast platform.

We identified several variability mechanisms used within KePlast in a former study [9]: *Module linking* is used during

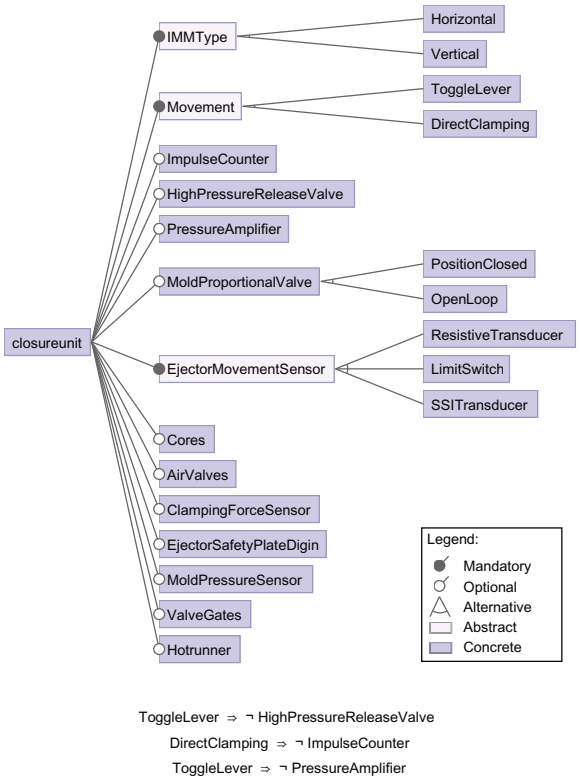

Fig. 3. Mold1 feature model.

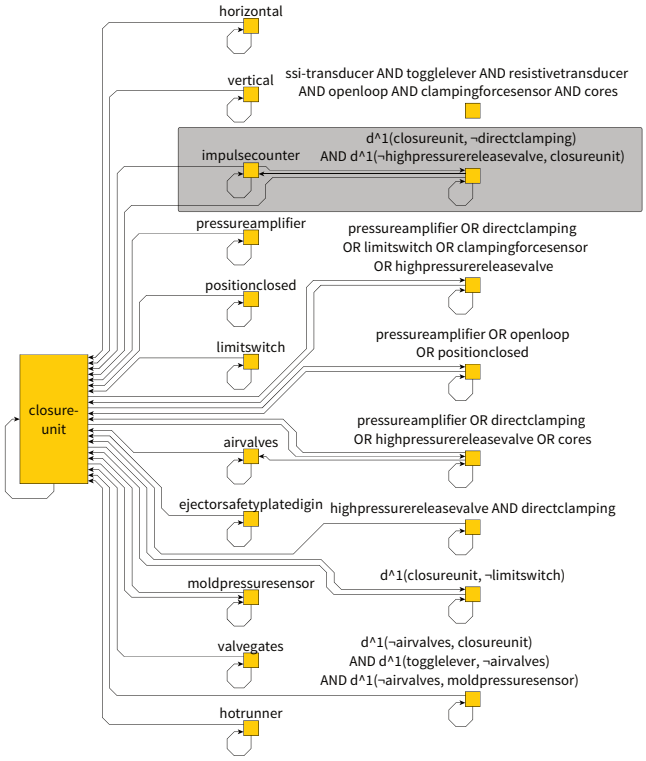

Fig. 4. Extracted trace dependency graph. Each node represents a trace to the code of a feature or feature interaction.

product customization. Different variants of a control function are available in different modules. Engineers decide which variant to use within a product by linking the respective module. *System variables* represent optional endpoints to machine equipment used in a product configuration. *Runtime configuration parameters* are available via the KePlast user interface for fine-grained configuration before system start-up.

Feature models have been developed to describe the variability of selected subsystems. The feature model for component Mold1 was reverse-engineered in two stages. First we analyzed Keba's custom-developed configuration tool, which is used for deriving a base system solution by selecting components and defining initial configuration settings. Our analysis resulted in an initial feature model for the component Mold1. Second we discussed and refined this initial feature model with the development lead of the KePlast platform. The feature model is shown in Figure 3.

B. Research Questions and Method

Does the extracted trace information correctly reflect the variability in the system? To answer this question we use ECCO's compositor which uses the extracted trace information to generate product variants. We use it to re-compose the same products that were used as input (i.e., the products with the same features) based solely on the extracted trace information. This means that we decide about including a trace's code based on the features of that product variant. We then compare these re-composed product variants to the corresponding original product variants generated by the KePlast product configuration tool.

More specifically, we create n KePlast product variants using the KePlast configurator and use them as input to our approach. The size of the KePlast products range from 53,000 to 58,000 lines of code when ignoring empty lines and comments. We then use the ECCO's compositor to re-compose these variants based on the extracted trace information. Finally, we compare the re-composed variants to the corresponding original variants generated by the KePlast configurator (after using CAPA to remove the dead code) and determine its correctness. Ideally they will all match their counterparts.

We measure the correctness of a composed product variant with respect to a ground-truth product variant generated from the KePlast configurator by means of the precision and recall metrics computed for the source code of the product variants. For example, a recall of 1.0 means that the composed product contains all the source code the original counterpart contains, and a precision of 1.0 means that the composed product contains no source code represented as AST nodes, that is not also contained in the original product variant.

When using $n = 15$ available KePlast product variants as input to our approach and then recomposing them using the extracted trace information, the resulting average precision is 1.0, i.e., the composed products contain no source code that is not also contained in their respective original product variants. The average recall is 1.0 which means that the composed products contain all source code of their original counterparts. This shows that the extracted traceability information is sufficient

to at least re-compose the original product variants, which we believe is a good indication that the extracted traces are correct. However, in future work we also need to determine precision and recall when composing yet unknown products variants. Furthermore, we need to evaluate our approach with respect to a ground truth provided by a developer.

Do the dependencies between the traces – the implementation variability – adhere to the design variability? To answer this question we compare the extracted trace dependency graph (i.e., the dependencies between the traces) to the feature model of the KePlast system and check whether the set of product variants described by the feature model is a subset of the set described by the dependency graph (i.e., allowed by the implementation). If the feature model would allow for the creation of product variants that are not supported by the implementation then possible reasons can be that i) the extracted traces and/or their dependencies are erroneous, ii) the previously reverse-engineered feature model is erroneous, or iii) we uncovered a flaw in the KePlast software system where it is possible to create variants that will not work. Specifically, we created n KePlast product variants and used them as input to our approach. We then compared the resulting dependency graph to the manually reverse-engineered feature model.

The extracted trace dependency graph is shown in Figure 4. The nodes are labeled with single features and feature interactions (written as δ^i(*feature*$_1$, ..., *feature*$_i$) with i representing the number of interacting features). Each node represents a trace to the code of a feature or feature interaction. Negative features ($\neg$*feature*) express that the features must not be present for the traced code to be included in a product variant. The dependency graph's structure is very simple. There are almost no dependencies between traces. Dependencies mostly occur within traces or with the *base*, as expected. One trace (in the top right) does not have any dependencies. This is because the corresponding features could not be associated with any code.

When comparing this dependency graph to the feature model shown in Figure 3 one can see high similarity. For the most part the dependency graph matches the feature model and the feature model violates only few dependencies in the graph. For example *closureunit* depends on *moldpressuresensor*. This would make the latter a mandatory feature which it should not be according to the feature model. Identifying the causes for these deviations will be part of our future work. For instance, we want to determine whether there is a flaw in the KePlast system, the reverse-engineered feature model, or the extracted traces. We hope that our approach will enable software engineers to efficiently and effectively identify and reconcile differences between how variability is modeled and how it is actually realized. Additionally one can see that the feature model's cross-tree constraint *DirectClamping* $\Rightarrow \neg$*ImpulseCounter*, saying that these features exclude each other, is also reflected in the dependency graph (see highlighted area in Figure 4). The trace for *ImpulseCounter* requires code elements from the trace containing $\neg$*DirectClamping* meaning that *ImpulseCounter* cannot be present in a product variant if also feature *DirectClamping* is present.

V. Conclusions and Future Work

Tools for recovering feature-to-code mappings, e.g. ECCO, in PLs have been shown to be useful, but they usually can handle only a single variability mechanism. Unfortunately, highly-configurable systems often require more than one variability mechanism. In this paper, we presented a new approach which combines the program analysis tool CAPA and the diffing tool ECCO to address the need to be able to handle mixed-variability systems. We also showed in our preliminary evaluation, that the approach produces sound output.

As future work we intend to develop a common technique for transforming between different kinds of PLs. As soon as software systems use more than one mechanism, any use of PL tools is difficult. This work is just a first step but already allows to handle a wide-spread combination of variability mechanisms, i.e., annotation-based and runtime variability approaches. Future work will also include investigations if converting between different mechanisms is possible without loosing information, e.g., the reverse engineering of SPLs by generating annotated static variability based on analyzing different variability mechanisms.

Acknowledgments

This work has been conducted in cooperation with KEBA AG, Austria, and was supported by the Christian Doppler Forschungsgesellschaft, Austria as well as the Austrian Science Fund (FWF) project P25289-N15 and Lise Meitner Fellowship M1421-N15.

References

[1] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *Proc. WCRE*, 2012, pp. 145–154.

[2] J. Rubin and M. Chechik, "Locating distinguishing features using diff sets," in *Proc. ASE*, 2012, pp. 242–245.

[3] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proc. OOPSLA*, 2011, pp. 805–824.

[4] C. Hammer, J. Krinke, and G. Snelting, "Information flow control for Java based on path conditions in dependence graphs," in *Proc. IEEE Int'l Symposium on Secure Software Engineering*, 2006, pp. 87–96.

[5] D. Lettner, F. Angerer, H. Prähofer, and P. Grünbacher, "A case study on software ecosystem characteristics in industrial automation software," in *Proc. Int'l Conf. on Software and Systems Process (ICSSP 2014)*, Nanjing, China, 2014.

[6] D. Lettner, F. Angerer, P. Grünbacher, and H. Prähofer, "Software evolution in an industrial automation ecosystem: An exploratory study," in *Proc. Int'l Euromicro Conf. on Software Engineering and Advanced Applications (SEAA 2014)*, Verona, Italy, 2014.

[7] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *Proc. ICSME*, 2014.

[8] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC*, T. Kishi, S. Jarzabek, and S. Gnesi, Eds. ACM, 2013, pp. 131–140.

[9] F. Angerer, H. Prähofer, D. Lettner, A. Grimmer, and P. Grünbacher, "Identifying inactive code in product lines with configuration-aware system dependence graphs," in *Proc. SPLC 2014*, Florence, Italy, 2014.

[10] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11–12, pp. 595–607, Dec. 1998.

[11] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," pp. 160–170, 1993.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration using feature models," in *Proc. SPLC*. Springer, 2004, pp. 266–283.

[13] M. Antkiewicz, W. Ji, K. Czarnecki, T. Berger, T. Schmorleiz, R. Laemmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *ICSE*, 2014.