Demonstration of a Toolchain for Feature Extraction, Analysis and Visualization on an Industrial Case Study

Sten Grüner, Andreas Burger, Hadil Abukwaik, Sascha El-Sharkawy, Klaus Schmid, Tewfik Ziadi, Anton Paule, Felix Suda, Alexander Viehl

Demonstration of a Toolchain for Feature Extraction, Analysis and Visualization on an Industrial Case Study

Sten Grüner*, Andreas Burger*, Hadil Abukwaik*, Sascha El-Sharkawy[†], Klaus Schmid[†], Tewfik Ziadi[‡], Anton Paule[§], Felix Suda[¶], Alexander Viehl[§]

*ABB Corporate Research Center Germany, Germany, {sten.gruener, andreas.burger, hadil.abukwaik}@de.abb.com

[†]Software Systems Engineering, University of Hildesheim, Germany, {elscha, schmid}@sse.uni-hildesheim.de

[‡]Sorbonne Université CNRS, LIP6 Paris, France, tewfik.ziadi@lip6.fr

[§]FZI Forschungszentrum Informatik, Germany, {anton.paule, alexander.viehl}@fzi.de

[¶]ScopeSET Technology GmbH, Germany, felix.suda@scopeset.de

***Abstract*—Transforming a clone-and-own (i.e. new product variants are created by copying and modifying existing artifacts) code structure and development process to a Software Product Line Engineering (PLE) approach is a tedious and error-prone task. Holistic tool support for such a process is highly desirable, especially to lower efforts and to speed up the transformation. Unfortunately, such a holistic toolchain for reverse engineering of variability, supporting variant-centric and platform-centric extraction approaches is not available. In this paper, we present a toolchain covering the first steps for moving a clone-and-own product development to a PLE approach. We validate the first prototype of the toolchain on a case study consisting of industrial firmware for smart motor controllers and we show that even this early prototype reduces time and effort for moving to a configurable platform approach in the sense of PLE.**

***Index Terms*—PLE, Feature Mining, Holistic Toolchain, Industrial Case Study**

I. Introduction

In the last three decades, Software Product Line Engineering (PLE) has become famous as an approach for reducing time-to-market of new products as well as development and maintenance costs of highly variable products. Additionally, the quality of the software increases by adopting PLE techniques as Fogdal et al. in [1] highlighted for a motor controller platform. In the academic field, there is plenty of knowledge regarding PLE and practices around it, such as the Product Line Hall of Fame [2], case studies, books or experiences presented at various PLE conferences. Nevertheless, adopting a specific PLE approach or pattern in industrial practice is not a trivial task and it requires a long adoption time.

One of the big drawbacks, when establishing a new product line approach in the context of existing software, is the need for *holistic tool support* covering all tasks needed to step-wise progress from a clone-and-own (i.e. new product variants are created by copying and modifying existing artifacts) development scheme to a PLE approach (e.g. as proposed in [**?**]). Such an integrated toolchain would lower the hurdle for starting the transformation process and give a company a starting point and the confidence for the migration. For sure, an integrated approach addresses a number of challenges like the step from clone-and-own to a product line strategy for new products, the discovery of hidden or not explicitly documented mechanisms to configure customer-driven variability, and the integration of an approach for selecting dedicated customer features per product variant. Facing and coping with these challenges, the aforementioned tool support rises confidence for moving to PLE approach on management level because the time for the transformation process may significantly be reduced. Likewise, the return-of-invest for adopting the PLE approach will be materialized earlier than before.

In this paper, we present the prototype of a holistic toolchain which supports a company in taking the first steps in the sketched migration process. The toolchain focuses on reverse engineering of variability by combining tools for analyzing an existing configurable software platform (platform-centric approach) with tools for analyzing currently available product variants (variant-centric approach). In order to get tangible and expressive results, the toolchain includes several visualizations of the results, coming from a management-oriented overview to dedicated technical feature-driven results. We performed a first validation of the toolchain based on an industrial firmware of smart motor controllers. The first results show the ability to reduce the time and effort for migrating a legacy code base into a PLE-ready code base.

The remainder of this paper is structured as follows: Section II discusses the state of the art and related work. Section III describes the used definition of software features, the industrial case study and the derived usage scenarios for the toolchain. Section IV introduces the toolchain and a typical workflow. Section V details the toolchain building blocks. In Section VI we present our initial evaluation results and close the paper with a summary and future work in Section VII.

II. Related Work

In this paper, we present a holistic toolchain for reverse engineering of variability to support the migration process of legacy software systems to a PLE approach. The technologies used for the presented analysis are not completely novel. However, the possibility of combination of different variability extraction approaches and tools is unique as no comparable toolchain for transformation and migration tasks exists today.

Various tools have been developed to analyze different aspects of variability information, like [3]–[7] for mining variability information from the Linux build system. Other tools extract the variability information from C/C++-based code files [8], [9]. Moreover, there are tools that detect variability-introduced dead code [10], family-based type checking and variable control-flow graphs [11], analysis of feature scattering across code files [12], or continuous on-the-fly identification of feature locations in committed code blocks [13]. While we reuse some of the extraction capabilities, we apply a completely different analysis as we reverse engineer variability information to introduce variability management.

In general, there exist two conceptually different strategies for reverse engineering of variability information of legacy systems: On the one hand, *variant-centric approaches* are used for mining variants to introduce a configurable platform. Thus, it is necessary to detect common and variable code parts and to introduce variability to make these code parts configurable. Examples are BUT4Reuse [14]. On the other hand, *platform-centric approaches* for analyzing undocumented variability in SPLs which was not available before. Examples here are, from Nadi et al. [15] which reverse engineer a variability model from code artifacts. This approach was extended for numerical features/conditions with fixed ranges in [16]. KernelHaven [17] used the extended approach to reverse engineer variability dependencies. In this work, the *bottom-up* and *top-down* approaches are combined to analyze a product line that uses cloning and configuration techniques to realize variants.

Most of the above-mentioned tools focus on one specific approach or aspect of the analysis and have limited possibilities to interface other tools enhancing their results. Here we present a holistic toolchain based on various multi-purpose tools to address the problem of product line migration in an integrated fashion.

III. Industrial Case Study

A. Feature Definition and Kinds of Feature Tracing

The definition of a “software feature” depends on the considered use case of the software product line [19]. After discussions with domain experts, we identified two types of features within the software system under consideration:

- *Front-end features.* This customer-facing type of features defines the functionality that is directly visible to an end-user and can, for example, be accessed via configuration tools of the product. These features provide direct customer value, therefore, their emergence and maintenance are in the focus of product management.
- *Back-end features.* This developer-facing type of features is usually not directly visible to the customer and they depend on (1) the selection of the front-end features and (2) the product-internal software and hardware architecture, e.g., the currently selected hardware platform, operational system, used compiler, etc. Typically, these features are maintained by the software development team.

Both feature types can be traced by using different techniques depending on the intended variability model, e.g., compile-time variability, link-time variability, or runtime variability. In our case, we differentiate between in-file-level (i.e., a subset of lines within a source code file) and file-level (i.e., a complete source file) feature annotations and we accordingly consider the following possibilities for their tracing:

- *File-level and in-file-level embedded annotations.* Storing the localization information along with the source code is the usage of embedded annotations. One possible format that we follow in the work was proposed in [20]. Combining feature annotations and source code allows storing and evolving both using the same tools and data sources, e.g., Integrated Development Environment (IDE) and Version Control System (VCS). The embedded annotations have no semantics in used programming language.
- *File-level tracing.* In case link-time variability mechanisms are used, the build chain may conditionally include specific source files/libraries that are linked into the compiled executable. The feature selection information is therefore stored in the build toolchain and its specific artifacts, e.g., Makefiles or configuration scripts.
- *In-file-level preprocessor-based tracing.* In case of utilization of some programming languages, e.g. C/C++, preprocessor directives can be used to mark source code blocks belonging to a specific software feature. This tracing type is a hybrid of the previous two types. In analogy to the annotation-based localization, it allows building a so-called 150% model containing all variant-specific implementation artifacts in one source file (e.g. a C/C++ source file with preprocessor directives). In contrast to embedded annotations, the precompiler flags have semantics within the programming language and can be configured by the build toolchain, i.e., the feature selection information is contained in build artifacts, while variability information is stored directly within the source files. Therefore, precompiler directives are usually mixed with file-level localization, s.t., a feature typically consists of files either fully included/excluded and source code blocks either activated/deactivated in the built variant.

In our case study and the feature analysis toolchain, we focus on locating and analyzing front-end features as well as finding dependencies between front-end and back-end features. Depending on the specific product family, a subset of those features has already been localized using file-level or even block-level precompiler based localization, other features are however not yet localized at all.

B. Case Study Description

For the case study, we use the firmware source code of a specific industrial motor controller (also called drive) product family consisting of about 1.2 MLoC of embedded C/C++ code. The code is compiled by a proprietary build chain to generate and compile the source code. Out of the one repository used for the case study, around twelve variants for different applications and hardware platforms of the motor controller can be compiled.

Therefore, the main non-functional requirements for the involved tools are the ability to parse C/C++ code, to process large code bases with >1 MLoC in a reasonable time and resource consumption, and to use/evaluate variability information that is contained in build toolchain artifacts.

In our previous work [18], we defined the process of feature extraction to consist of *feature identification* (i.e., defining the feature name and its functionality), *feature localization* (i.e., mapping the feature to software engineering artifacts like source code blocks or files), and *feature tracing* (i.e., storing and documenting the localization information).

C. Usage Scenarios for the Feature Toolchain

Based on the feature tracing techniques that we introduced in Section III-A, we define the following application scenarios for the proposed toolchain:

S1: Feature extraction without considering existing variant information. This scenario addresses the extraction of features out of unannotated source code without considering existing product variants. The advantage of such a scenario is its simplicity. That is, available pattern matching and Information Retrieval (IR) based tools might work on pure source code without build-specific information. However, it will probably be needed for precise extraction of information from a specific build toolchain (e.g., a compilation database in [18]).

S2: Feature extraction combined with existing variant information. This scenario aims to extract features by comparing available variants, e.g., firmware variants. Such a comparison requires the extraction of variability information that is stored within the build toolchain, e.g., at least a list of compiled source files. Extraction scenarios S1 and S2 can also be combined, i.e., the features are first extracted without variant information and are cross-validated using variant information in a second step. Furthermore, considering built variants yields a mapping of available features to a specific variant.

S3: Feature visualization and metrics. In order to assess, to refine, and to monitor extracted features, there is a need for tools providing a user-friendly interface for feature annotations. Typical use cases are the visualization of feature locations across the build artifacts and the visualization of dependencies between different features for continuous feature tracing by software developers. Furthermore, a calculation of metrics on the level of features, files and folders (cf. Table 1 in [21]) is useful to monitor and accompany the feature evolution along the software development process.

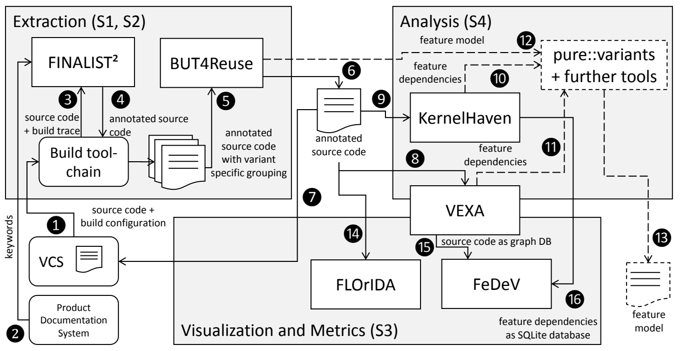

Fig. 1. Toolchain Demonstrator Overview and Covered Usage Scenarios

S4: Feature analysis. The analysis part of the toolchain is responsible for processing available feature localization information within the source code to discover feature dependencies and propose a feature hierarchy. One of the challenges is the consideration of the interplay between embedded annotations and preprocessor directives due to their different semantics. Ultimately, the discovered dependencies and presence conditions for features can be used as input to create a feature model, e.g., a feature tree.

IV. Toolchain Overview

In this section, we present an overview of the toolchain demonstrator for feature extraction, analysis, and visualization. We briefly describe a typical interaction sequence within the toolchain following the arrow numbers in Fig. 1 where arrows depict data flow between the tools and dashed lines represent planned, but not yet implemented functionality. The details for the invoked tools are provided in Section V.

All feature extraction scenarios described in Section III-C start with checking out the unannotated source code from a repository within the VCS into a location accessible by the build toolchain (arrow ❶). This code contains some precompiler-based defines, but no embedded annotations.

For the specific feature extraction without considering existing variant information scenario, a set of keywords is manually extracted from the device product documentation that describes the user-facing features (arrow ❷). Furthermore, the code is compiled (and especially preprocessed with C/C++ compiler) by the build toolchain for one variant in order to get a consistent code base (arrow ❸). The results of compilation manifest in a build trace, a Clang compilation database, which contains the list of compiler invocations, i.e., the list of compiled source code files and active precompiler definitions. The build trace and the source code base are used for static code analysis by FINALIST^2. It documents localized features by embedded annotations within the source code.

For the specific feature extraction with considering existing variant information scenario, the annotated code (arrow ❹) is passed to the build toolchain that builds as many existing product variants as possible. The traces of the build chain are post-processed, s.t., only actually compiled files are sorted into the variant-specific folders. At this point, embedded annotations coming from the previous step are possibly replicated if some product variants share common source files. Folders containing relevant source files are now provided to BUT4Reuse (arrow ❺), which extracts the similarities and differences between variants (details follow in the subsequent section). The extracted information is embedded into source code using annotations (arrow ❻, currently done manually).

Thus, the first two scenarios related to feature extraction that were mentioned in Section III-C are finished, and the extracted feature data can be saved back into the VCS (arrow ❼).

The feature analysis scenario starts by importing the annotated source code artifacts into the respective tools: VEXA (arrow ❽) and KernelHaven (arrow ❾). The extracted feature dependency information is supposed to be used for the construction of a feature model (arrows ❿ and arrow ⓫). Furthermore, feature dependency information extracted by BUT4Reuse can be included (arrow ⓬). We aim to combine feature dependency information in a feature model (arrow ⓭), e.g., as a pure::variants model.

Tools that are relevant for scenario S3 can operate on plain annotated source code as in case of FLOrIDA (arrow ⓮). Another possibility is to reuse information that is extracted by other tools as done by FeDeV. That is, it uses VEXA output containing the representation of the annotated source (arrow ⓯). Furthermore, it uses KernelHaven output that can be processed in the format of SQLite databases (arrow ⓰).

We expect an iterative usage of the toolchain, i.e., the represented sequence of tool invocations can be (partially) repeated covering several usage scenarios from Section III-C. One example is an iterative refinement of extracted software features following arrows ❶ to ❼ as described in [18].

V. Tool Overview

Here, we provide an overview of the individual building blocks of the demonstrator toolchain. We describe their functionality based on their respective input and output artifacts.

Many tools aim to provide an integrated solution covering multiple scenarios listed in Section III-C. Below, we list only the subset of tool functionalities that are currently used within the toolchain demonstrator (cf. solid lines in Fig. 1). Extending usage of the tool to cover their full functionality is part of future work (cf. dashed lines in Fig. 1).

A. FINALIsT^2

1) Input artifacts: Unannotated C/C++ code, list of keywords used for IR to localize the feature.

2) Functionality and user interaction: Assisted iterative feature extraction based on IR and static code analysis.

3) Output artifacts: Annotated C/C++ code.

4) Tool summary: The Feature Identification, Localization, and Tracing Tool (FINALIsT^2) [18] supports developers by identifying, locating and documenting features within the source code. It uses an iterative semi-automated approach for feature mining by combining IR techniques and static code analysis. Within the sketched demonstrator in Fig. 1, FINALIsT^2 is used to assist developer or domain experts to identify and document features. Therefore, the tool uses an IR approach to identify a first source or header file as a starting point for the static code analysis. The IR search engine, Apache Lucene, needs a query, which can be extracted for example from a firmware manual or the user can define the query by himself. Once the starting point is identified, FINALIsT^2 starts a static code analysis by exploring incoming and outgoing dependencies of the current cluster (i.e., function calls, variable access, and file inclusion dependencies). The cluster can be then iteratively refined by the user.

B. BUT4Reuse

1) Input artifacts: Variant artifacts, e.g., annotated or unannotated C/C++ code for each variant.

2) Functionality and user interaction: Graphical overview and summary of features contained in each variant (feature list). Feature constraints (currently not used in the toolchain).

3) Output artifacts: Annotated C/C++ code containing annotations for discovered features (currently done manually).

4) Tool summary: Bottom-Up Technologies for Reuse (BUT4Reuse) [14], [22] is a generic and extensible framework implementing the different activities related to variability engineering. This mainly includes feature identification and naming, feature location, constraint discovering, and feature model synthesis. BUT4Reuse takes as input a collection of artifact variants and apply different algorithms for the mentioned activities. In this work, we used BUT4Reuse for the following two activities: 1) feature identification and naming where the objective is to analyze the input variants and identify the implementation fragments that implement the existing features. Naming aims to propose solutions for domain experts to name the identified features. 2) Feature model synthesis that extracts an organization of the identified features following the feature model notations.

To support each artifact type, BUT4Reuse uses the concept of *adapter* that processes the input artifact to be used by the different algorithms. In this paper and while we are considering the C/C++ source code of a collection of variants, we used a BUT4Reuse specific adapter that allows comparing the C/C++ source code of input variants. In addition to the different algorithms, BUT4Reuse is based on a set of visualization paradigms that guide the feature identification process. As shown in the next section, this visualization is useful for domain experts to name the identified feature and understand the variability inside the analyzed variants.

Summarizing the description of extraction tools (FINALIsT^2 and BUT4Reuse), it is worth mentioning that their performance was evaluated in terms of accuracy of identifying feature locations in Section VI-A.

C. VEXA

1) Input artifacts: Annotated C/C++ code containing embedded annotations and precompiler statements.

2) Functionality and user interaction: VEXA's functionality is invoked through Cypher queries, which can be executed interactively one at a time or as a complete analysis batch containing a multitude of queries. Interaction with the user is realized through a web front-end with visualization capabilities for presenting various aspects of the preprocessed source code artifacts, e.g., feature metrics and dependencies between preprocessor statements and embedded annotations.

3) Output artifacts: Graph database which stores the property graph that contains all feature localization information; discovered dependencies between preprocessor directives and embedded annotations.

4) Tool summary: The Variability Extraction and Analysis (VEXA) toolkit is a versatile collection of complementary procedures to help with many different tasks of variability extraction, feature analysis, visualization, and the calculation of user-defined metrics. Implemented as a plug-in for the "world's leading Graph Database" Neo4j[1], the VEXA toolkit leverages the powerful graph storage and processing capabilities of Neo4j to enable detailed dependency analyses of source code artifacts (e.g., #ifdef variability in C/C++ code) and reveal intricate feature connections across project artifacts along with graph visualization possibilities.

Tool's architecture rests upon two main pillars:

- *The Property Graph Model.* It is represented by a set of nodes and relationships that can both hold any number of attributes (key-value pairs) called *properties*. Nodes are entities in the graph that can be tagged with labels. Relationships always have a direction, a type, a start node, and an end node. They provide directed, named and semantically-relevant connections between two nodes.
- *The Cypher Query Language*[2]. Cypher is a declarative query language that allows for expressive querying and efficient manipulating of a property graph. The VEXA toolkit extends Cypher using the concept of user-defined procedures and functions to provide custom implementations of extraction and analysis procedures, which then can be called from Cypher directly.

The methodology behind VEXA pursues an iterative workflow, in which the user of the toolkit (a software developer or domain expert) specifies all analysis steps in a flexible way—tailored to his specific needs—using Cypher queries.

Context-specific Cypher queries can be employed to support the user in visualizing relevant dependencies between analyzed artifacts, e.g. feature locations and their relation to source files and code elements, in the form of graphs and tables.

D. KernelHaven

1) Input artifacts: Annotated C/C++ code containing embedded annotations and precompiler statements.

2) Functionality and user interaction: Extraction of variability based on in-file annotations, no user interaction.

3) Output artifacts: Excel file and SQLite database containing relations between identified features.

4) Tool summary: KernelHaven[3] [17] is an experimentation workbench designed to simplify the realization of various analyses in the domain of static software product line analysis. Within the demonstrator, KernelHaven is used to extract variability information from C-preprocessor statements and annotations of C/C++ files and perform further analysis on this information. Thus, KernelHaven is used for the analysis of annotations as created by the FINALIsT[2] tool (cf. Section V-A) and optionally detects inconsistent, manually-written annotations. KernelHaven also provides a feature effect analysis as described in [15] to compute variability dependency information among features from annotations and C-preprocessor statements. The result of the analysis is a Boolean precondition for each feature used in a code artifact, showing when its selection influences the product derivation.

A subsequent analysis component uses the dependency information to compute a dependency graph among the features. The graph contains directed edges between features and their dependent features. Annotations of the edges show whether the depending feature must not be selected (e.g., *exclusive* relationship), must be selected (e.g., *requires* relationship), or a propositional formula, if the precondition of the feature effect analysis cannot be split into one of the two previous cases.

E. FLOrIDA

1) Input artifacts: Annotated C/C++ code.

2) Functionality and user interaction: Graphical user interface to visualize and navigate through features that are scattered across the file system.

3) Output artifacts: -

4) Tool summary: The Feature Location and Identification Dashboard (FLOrIDA) is used within the demonstrator for visualizing features, their metrics, and their dependencies. The tool has been thoughtfully discussed in [21].

F. FeDeV

1) Input artifacts: Graph database originating from VEXA tool or SQLite database originating from KernelHaven.

2) Functionality and user interaction: Interactive visualization and navigation of feature dependencies, feature annotations, and variation points within source files.

3) Output artifacts: Visualization of feature dependencies, variation points, and file systems as tables, graphs, and trees.

4) Tool summary: The Feature Dependency Visualization (FeDeV) tool is a visualization application for analysis results of KernelHaven and VEXA based on the TomSawyer Visualization framework[4]. The tool covers tabular lists for all data elements, some tree structures (e.g., to represent variation points within files, or showing a directory hierarchy) and interactive graphs to inspect the analysis results. The tool mainly focuses on visualizing the feature dependencies, feature annotations, and variation points. Each analysis source is mapped to an internal metamodel and instantiated during

[1]https://github.com/neo4j/neo4j
[2]https://www.opencypher.org/
[3]Publicly available at https://github.com/KernelHaven/KernelHaven
[4]https://www.tomsawyer.com/products/visualization/

an import process. Currently, FeDeV is able to import, inspect and explore the analysis results of KernelHaven and VEXA.

The inspection workflow depends on the desired analysis. Dependencies of features are explored step-by-step. Starting with an empty graph, identified features can be added via table selection or a search function and composed to a feature-dependency-graph. There are also dedicated commands to add features of incoming and outgoing dependencies to the graph as well as a path of dependencies up to related root features (i.e., independent features) of the selected context feature. In contrast, variation points and feature annotations detected by the VEXA can be inspected by browsing the analyzed file tree or a projection of feature annotations used by files.

A feature is represented as a node, showing its name, the number of incoming and outgoing dependencies, and a textual constraint (if defined). Dependencies are denoted as edges between feature nodes. Each dependency can have a constraint fragment, that is calculated from the actual feature's constraint and describes, which additional features must be selected to get a valid variant configuration.

VI. Preliminary Evaluation Results

In the following, the preliminary results of the toolchain evaluation are discussed. Tools of the first workflow stage of feature extraction (scenarios S1 and S2) have been evaluated more thoughtfully as a foundation for the following steps.

A. Feature Extraction

1) FINALiST^2: In previous work [18], we performed no expert interviews and were able to achieve 98% accuracy on file-level (full files included in a feature) and 68% accuracy on in-file-level (certain lines of a file included in a feature) for a feature when compared to included precompiler definitions.

For this work, we performed an initial comparison of unassisted and tool-supported extraction of one randomly picked front-end and one back-end feature. The extraction was performed by non-experts in drive firmware development and cross-checked in expert interviews. The tool-assisted extraction was measured to be 69% quicker than comparable manual extraction for both features having a similar accuracy as confirmed by the development team.

2) BUT4Reuse: On the code base with 12 variants BUT4Reuse identified the scattered features that were covered as "blocks" with temporary IDs and represented them as a variant-feature matrix (see Fig. 2) where block "A" represents the code that is shared by every variant. Word cloud feature helped in recognizing and renaming the identified blocks.

In the code base, the tool identified 24 features out of around 80 front-end features. Worth to mention, the tool only finds differences between built variants, s.t., a feature identified by the tool typically contains multiple features as defined in Section III-A. Therefore, the generated insights are considered very useful and are used to validate/extend the extraction results from the FINALiST^2 tool as no additional information for feature identification like keywords is needed.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
V1	X	X	X	X	X	X	X	X	X		X	X		X		X	X	X						
V2	X	X	X	X	X	X	X	X		X	X	X	X											
V3	X	X	X	X	X	X	X	X	X	X	X		X											
V4	X	X	X	X	X	X	X	X	X	X	X	X	X	X										
V5	X	X	X	X		X	X	X		X		X	X		X						X			
V6	X	X		X	X				X					X	X		X		X			X		
V7	X	X	X		X	X		X	X	X	X	X				X							X	
V8	X		X	X	X		X	X	X	X			X	X	X				X					X
V9	X	X	X	X	X	X	X		X	X		X	X		X					X				
V10	X	X	X	X	X	X	X		X	X			X		X					X				
V11	X	X	X	X	X	X	X	X			X	X				X	X	X						
V12	X	X	X	X	X	X	X	X	X		X			X		X	X	X						

Fig. 2. BUT4Reuse: The variant-feature matrix (names are obfuscated)

B. Feature Analysis

1) VEXA: Feature analysis is performed by writing custom Cypher queries. In brevity, we present a small part of the overall analysis to demonstrate what is possible. After VEXA processed the code base, the property graph (a graph representation of the whole code base containing all embedded annotations, preprocessor directives, code statements, source files, etc.) is available and can be queried for information.

The first task was listing source code files containing specific feature annotations.The second task was defining a query to show which feature annotations contain which precompiler flags to find out and compare eventually double-annotations.

With the help of a Cypher expert, we were able to create queries to solve both tasks. The queries required intermediate steps that inserted additional information into the graph database. On one hand, it proved the flexibility of the tool. On the other hand, a good knowledge of Cypher language and VEXA property graph structure is needed to use it efficiently.

2) KernelHaven: By using KernelHaven we were able to identify multiple inconsistencies in manually injected code annotations, e.g., forgotten closing block annotation or typos. Furthermore, a specific dependency between a preprocessor-#ifdef and an embedded annotation was found and manually verified by tracing through the code base. The dependency stated that the disabling of the preprocessor statement removes all feature-annotated code blocks. A larger evaluation of feature dependencies in the code base is currently pending.

C. Feature Visualization and Metrics

1) FLOrIDA: Visualization possibilities of the FLOrIDA tool were discussed in [21].

2) FeDeV: We evaluated FeDeV on the output of both VEXA and KernelHaven. FeDeV provides different views like tables, trees, and graphs for manual inspection of extraction results. To simplify the visualization of large code bases, FeDeV provides a useful feature of predefined and user-defined filters to hide non-relevant features from such graphs.

Regarding VEXA results on the feature annotation and #ifdef block level, the coupling of annotated features and files is visualized graphically allowing to inspect the number of annotated code blocks and their impact on the source code file. Different layout options allow to re-center from feature- to file-centric view easily. Results of KernelHaven analysis are rendered as dependency feature graphs providing color coding of edges and links provide additional information, e.g., to identify mandatory and optional features visually.

VII. Summary and Future Work

In this paper, we presented a prototype of a holistic toolchain for feature extraction, analysis, and visualization that combines variant-centric and platform-centric tools for reverse engineering of variability information. We were able to show first promising improvements in terms of reduced time and effort by iterative usage of the toolchain for feature extraction.

Future work can be separated into two dimensions: On one hand, improving the evaluation methodology by defining quantitative KPIs and performing a more reliable evaluation of the used case study. Furthermore, the case study can be extended to consider multiple clone-and-own repositories, e.g., to locate similar/same features across product groups to automatically generate 150% model files and eventually merge these into one.

On the other hand, the toolchain itself, single tool building blocks and their inputs/outputs can be improved, in particular:

- Improving precision of variant-centric feature extraction by pre-processing source files before analysis by means of a compiler or further tools, e.g., Coan[5].
- Use and combine constraint information from variant-centric extraction with feature constraints and relations.
- Further steps towards a complete feature model extraction, e.g., by generating pure::variants feature models.
- Merge tool results in one visualization view of feature location and dependencies between them.
- A unified graphical or command-line interface.

Acknowledgments

This work is supported by the ITEA 3 15010 project REVaMP[2] funded by the German Federal Ministry of Education and Research (01IS16042A, 01IS16042B, 01IS16042F, 01IS16042H) as well as FUI the Ile-de-France region and BPI in France.

References

[1] T. Fogdal, H. Scherrebeck, J. Kuusela, M. Becker, and B. Zhang, "Ten years of product line engineering at danfoss: lessons learned and way ahead," in *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 2016, pp. 252–261.

[2] D. M. Weiss, P. C. Clements, K. Kang, and C. Krueger, "Software product line hall of fame," in *Software Product Line Conference, 2006 10th International*. IEEE, 2006, pp. 237–237.

[3] S. Nadi and R. Holt, "Mining kbuild to detect variability anomalies in Linux," in *16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012, pp. 107–116.

[4] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the linux build system," in *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, vol. 1, 2012, pp. 21–30.

[5] T. Berger and C. Kästner, "KBuildMiner," 2016, last visited: 29.01.2019. [Online]. Available: https://github.com/ckaestne/KBuildMiner

[6] J. Sincero, R. Tartler, C. Egger, W. Schröder-Preikschat, and D. Lohmann, "Facing the linux 8000 feature nightmare," in *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010.

[7] C. Kästner, "KconfigReader," 2016, last visited: 29.01.2019. [Online]. Available: https://github.com/ckaestne/kconfigreader

[8] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat, "Efficient extraction and analysis of preprocessor-based variability," in *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*, 2010.

[9] "TypeChef," 2017, last visited: 29.01.2019. [Online]. Available: https://github.com/ckaestne/TypeChef

[10] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem," in *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*, 2011, pp. 47–60.

[11] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, 2013, pp. 81–91.

[12] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, 2018, (Early Access).

[13] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 529–533.

[14] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon, "Bottom-up technologies for reuse: Automated extractive adoption of software product lines," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, may 2017. [Online]. Available: https://doi.org/10.1109/icse-c.2017.15

[15] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, Aug 2015.

[16] A. Krafczyk, S. El-Sharkawy, and K. Schmid, "Reverse engineering code dependencies: Converting integer-based variability to propositional logic," in *Proceeedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, vol. 2, 2018, pp. 34–41.

[17] C. Kröher, S. El-Sharkawy, and K. Schmid, "Kernelhaven - an experimentation workbench for analyzing software product lines," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18): Companion Proceedings*, 2018, pp. 73–76.

[18] A. Burger and S. Grüner, "Finalist 2: Feature identification, localization, and tracing tool," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 532–537.

[19] D. Beuche and M. Dalgarno, "Software product line engineering with feature models," *Overload Journal*, vol. 78, pp. 5–8, 2007.

[20] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining feature traceability with embedded annotations," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 61–70.

[21] B. Andam, A. Burger, T. Berger, and M. R. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2017, pp. 100–107.

[22] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up adoption of software product lines: a generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, D. C. Schmidt, Ed. ACM, 2015, pp. 101–110. [Online]. Available: https://doi.org/10.1145/2791060.2791086

[5]http://coan2.sourceforge.net/