An Overview on Analysis Tools for Software Product Lines

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Gunter Saake
University of Magdeburg, Germany

ABSTRACT

A software product line is a set of different software products that share commonalities. For a selection of features, specialized products of one domain can be generated automatically from domain artifacts. However, analyses of software product lines need to handle a large number of products that can be exponential in the number of features. In the last decade, many approaches have been proposed to analyze software product lines efficiently. For some of these approaches tool support is available. Based on a recent survey on analysis for software product lines, we provide a first overview on such tools. While our discussion is limited to analysis tools, we provide an accompanying website covering further tools for product-line development. We compare tools according to their analysis and implementation strategy to identify underrepresented areas. In addition, we want to ease the reuse of existing tools for researchers and students, and to simplify research transfer to practice.

Keywords

Software product lines, tool support, sampling, testing, type checking, static analysis, model checking, theorem proving, non-functional properties, code metrics

1. INTRODUCTION

A *software product line* is a set of different software systems that share commonalities, described by means of features [16]. A *feature* is a user-visible aspect or characteristic of a system [26]. *Feature-oriented software development* (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems [3]. In FOSD, a product of a software product line is generated automatically for a selection of features. FOSD facilitates the construction of customized software products, while artifacts can be reused and thereby time and resources can be saved.

Customizable software is necessary for a broad spectrum of domains (e.g., operating systems for diverse hardware). However, just like single systems, software product lines need to be analyzed (e.g., for type safety). Because analysis tools from single system engineering can only be applied to one product at once, these tools are not sufficient for an efficient analysis of all products (e.g., to ensure type safety of all products). In the last decade, several approaches have been proposed that transfer the analysis of single systems to an efficient analysis of product lines. With these strategies there also came tools that can be applied to analyze a product line. To get a representative list of current tools, we base this overview on a recent survey on analyses for software product lines [68]. However, for brevity, we focus on the analysis of source code and byte code (i.e., analysis of software models and variability models is excluded). Furthermore, because of the high number of tools and the limited space in this paper, we cannot discuss the approaches of each tool in detail.

With this paper and our website, we aim to help researchers, students, and practitioners working in the field of FOSD. According to our experience, the effort of building a research prototype or creating a proof of concept is greatly reduced if existing tools can be leveraged. It is also important to know which tools exist and on which other tools they are built on, which can also reduce the effort of building a new tool. This work provides the following contributions:

- A brief overview on distinguishing characteristics of product-line analysis tools
- An overview on analysis tools
- We maintain a website for product-line tools in general, because new tools will be developed in future.[1]

Our goal with this workshop is to discuss characteristics of analysis tools and identification of tools that could be added to our website.

The paper is structured as follows. In Section 2, we briefly introduce the background of FOSD and characteristics of product-line analysis tools. We present tools for testing in Section 3, tools for verification in Section 4, and further analysis tools in Section 5. We summarize our results in Section 6 and conclude in Section 7.

2. CHARACTERISTICS OF PRODUCT-LINE TOOLS

FOSD is the process of developing software product lines in terms of their features. According to Apel et al. [3], FOSD can be divided into the four phases: (1) domain analysis, (2) domain design and specification, (3) domain im-

[1] www.fosd.net/tools

plementation, and (4) product configuration and generation. In domain analysis, commonalities and differences of the domain of interest are identified, resulting in a feature model [26]. Domain design and specification is the process in which the architecture of the product line is designed and specified. Domain implementation is the phase of designing, implementing, analyzing, and refactoring the source code of the product line. In product configuration and generation, a product containing a desired selection of features is created. In our work, we focus on tools for software analysis of product lines. For domain analysis we refer to the survey of Benavides et al. [10] about approaches for analysis of feature models, and the survey of Lisboa et al. [40] about tools for domain analysis. Because we focus on tools, we can only give brief introductions to the aspects of FOSD, thus we refer to previous works that give a general overview on FOSD [3], and analysis [68] thereof.

In this overview, we present tools for product-line analysis. To ease the choice of an appropriate tool, we categorize the tools with respect to four dimensions, namely, product-line implementation technique, analysis technique, strategies for product-line analysis, and strategy of the tool.

Product-Line Implementation Techniques. Implementation techniques for product lines can be divided into annotation-based and composition-based approaches [3]. With annotation-based approaches, code segments can be annotated with a feature or a feature expression, and then activated or deactivated depending on the selection of the features. Annotation-based approaches include preprocessors and virtual separation of concerns [27]. Products from composition-based product lines are composed out of a set of composable units [3]. Composition-based approaches include feature-oriented programming [52], aspect-oriented programming [31], and delta-oriented programming [58]. For brevity, we distinguish only between tools for annotation-based and composition-based product lines. However, we classify the tools into the specific implementation approach on our website.

Software Analyses. In this paper we survey tools for different analysis techniques. To simplify the choice of an appropriate tool, we differentiate the analysis techniques into three categories; testing, verification, and further analyses. In the category of testing, we collect tools that execute programs during analyses. In the category verification, we collect tools that analyze the validity of the program without a need to execute the program. Finally, we collect further analysis tools that do not match in the previous groups, such as code metrics.

Strategies for Product-Line Analysis. Software analyses are crucial for the efficient development of software. However, applying mechanisms and tools for analysis of single systems on generated products involves redundant analyses and is not feasible for large product lines [68]. More efficient approaches have been developed for analysis of software product lines, which avoid the generation and individual analysis of all products. For our overview we reuse the categories for product-line analysis of a recent survey [68]. One strategy is to generate representative products that cover as many errors as possible; this is called optimized product-based analysis. Generally, analyses that are based on generated products are called product-based. Approaches that take variability into account and do not need to generate all products are called family-based. Approaches that analyze the artifacts of each feature in isolation are called feature-based analyses. Furthermore, combinations of the analysis strategies are possible (e.g., feature-family-based).

Strategy of the Tool. The goal of a tool is to implement a software analysis with a specific product-line analysis strategy. To apply an analysis to product-lines, there are three main strategies. First, the tool uses *product-based* analysis where the actual analysis is done on generated products. Second, the tool is *variability-aware* and can handle the product line as-is (e.g., the tool considers variability defined with `#ifdef` statements). The third strategy is to use *variability encoding* (a.k.a. configuration lifting) [51]. With variability encoding, compile-time variability is translated into run-time variability (e.g., for preprocessors: `#ifdef FEATURE` is translated into `if (FEATURE)`). The result of variability encoding is a metaproduct (a.k.a. product simulator) which can simulate all products using a feature selection as input [51]. Using such a metaproduct, existing tools that can handle run-time variability (e.g., model checker) can be applied to efficiently analyze the product line.

In the following Sections 3, 4, and 5, we give overview on existing tools for product-line analysis. Based on this overview, we discuss the current state of tool support in Section 6 (e.g, which parts are well or not supported).

3. PRODUCT-LINE TESTING

Testing is an analysis technique that needs to execute the program to analyze a given property. In this section, we first cover sampling tools. These tools support testing by generating a representative subset of products. In the second part, we cover tools that support testing of product lines beyond sampling.

Sampling. In sample-based analyses, a representative subset of all products for a given coverage criterion is analyzed [3, 68]. With this approach, it is possible to efficiently detect errors by applying analysis tools from single-system engineering. Sampling is often used in combination with testing, but can be applied to verification and other analyses as well [8, 24, 39, 50, 68]. Sampling strategies are sound but always incomplete, since they can only detect errors that are contained in the subset that is analyzed [3].

A popular sampling strategy is T-wise coverage which aims to detect interactions of T features, because feature interactions are likely to cause errors for few features (e.g., pairwise coverage for T = 2) [3]. Sampling for T = 2 is implemented by, MoSo-PoLiTe [46] integrated in the configuration management tool pure::variants [53], Pacogen [23] which can generate a minimal-sized set of configurations, and Perrouin et al. who developed a scalable toolset for T-wise coverage using Alloy [48]. SPLCATool [25] supports sampling with several algorithms, such as ICPL (T $\in$ {1,...,3}), MoSo-PoLiTe (T $\in$ {2,...,3}), Chvatal (T $\in$ {1,...,4}), IPOG (T $\in$ {1,...,6}), and CASA (T $\in$ {1,...,6}). T-wise testing for product lines is a special case of combinatorial testing [44]. For an overview on conventional tools for combinatorial testing, we refer to the website of Jacek Czerwonka.[2]

The Undertaker [66] tool can generate Linux configurations for a given source file to achieve nearly full configurati-

[2] `www.pairwise.org`

on coverage [65] (aka. feature coverage [3]). With configuration coverage, each feature has to be selected at least once. In contrast, at T-wise sampling with T = 1 each feature has to be unselected at least once additionally. The tools PLEDGE [22] and GeneticTestCaseGeneration [18] use genetic algorithms to generate products. There are more sampling strategies than T-wise and feature coverage. However, because tools for such approaches are missing, we do not discuss these approaches here and refer to Apel et al. [3].

Testing. Similar to testing of single systems, software product line testing aims to uncover defects, however, with additional management of variability [47]. The main challenges of testing software product lines are reusing test cases, and reducing redundancies in test cases and executions. However, in this section we only mention tools that do not focus on sampling, because they are already covered under sampling. Because we focus on tools, we refer to recent surveys [17, 13] for more information on product-line testing. We discovered three categories of tools for product-line testing namely, test-case generation, product reduction, and family-based testing. However, these categories should not be complementary to the main categories for product-line analysis.

Tools for *test-case generation* of product lines automatically generate customized test cases for a given product. The Generative Aspect-oriented Testing framework (GATE) [19] can generate unit tests out of a unit test case repository. The tool Kesit [73] uses AHEAD [9] and SAT-based analysis to automatically generate test inputs for each product in a product line. Asadal [35] and the Mobile Application Test Environment (MATE) [54] provide model-based testing for product lines. The tool ParTeG [75] generates unit tests automatically based on model-based testing.

Tools for *product reduction* reduce the number of products to test for a specific test case. Shi et al. [60] developed a prototype for compositional symbolic execution (denoted by us as cse) that works in concert with a feature dependence graph to reduce feature interactions that must be tested. SharQ [74] is a framework of projects dealing with hardware and software testing for FOP. SPLTester [32] is a tool that reduces the combinatorial number of programs to test by determining behavior-irrelevant features for a given test case. Stricker et al. [64] developed a prototype for their model-based technique ScenTED-DF which avoids redundant testing in application engineering. The tool splmonitor [33], solves the problem of run-time monitoring of annotation-based product-lines for a safety-property, by reducing the set of possible programs by identifying variants where the property can never be violated.

Family-based testing tools can execute all products in parallel for a given test case saving intermediate states. A tool for parallel testing based on the interpreter from JavaPathfinder [21] is provided by Kim et al. (denoted by us as shared-execution) [34]. Kästner et al. [30] developed a prototype interpreter (denoted by us as vai) for the WHILE language which stores different values for variables depending on selected features at the same time. Similar to this, they developed the tool Varex [43], a variability-aware interpreter for PHP that performs computation in multi-value data for testing of plug-in-based web applications.

4. PRODUCT-LINE VERIFICATION

In this section, we discuss tools for product-line verification. We categorize the tools according to the analysis technique scaled to product lines, such as type checking.

Type Checking. The basis of most approaches that analyze software product lines is type safety of all products, such that each product can be compiled. Type checking is the verification process that ensures type safety [49]. Efficient type checking tools for software product lines consider the variability defined at the feature model for family-based analyses based on a unmodified product line.

Type checker for annotation-based product-lines consider variability defined at annotations and at the feature model to reason about type safety. The tool TypeChef [29] is a type checker for the C preprocessor that is able to check all configurations of the Linux kernel. CIDE [27] provides a product-line-aware type system for virtual separation of concerns.

Composition-based type-checking considers the dependencies between feature modules to ensure type safety after composition. The tool Fuji [5] provides product-based, feature-based, and family-based type checking of feature-oriented product-lines in Java. The tool FeatureTweezer [6] provides a generalized approach that provides language-independent type checks of FeatureHouse [4] product lines. SafeComp [9] is a tool for type-safe generation of products using a variability-aware type system available with the AHEAD Tool Suite [9]. DeltaJ [58] provides family-based type checks for delta-oriented programming.

Static Analysis. Static analysis operates on compile-time and can predict dynamic values or behaviors that arise at run-time [45] (e.g., to avoid superfluous computations of unused values). Bodden et al. provide the static analysis tool SPL^{LIFT} [12], which provides automatic family-based analysis based on the IFDS framework [55] and CIDE [27] for virtual separation of concerns. The tool Emergo [56] supports developers maintaining annotation-based product lines using contracts between features, such that other features cannot be broken. The Aspect Composition Validation tool (ACVTool) [36] provides automate static analysis of feature dependencies for aspect-oriented programming using design by contract [42]. SpeK [59] is a tool for automated detection of feature interactions in feature-oriented product-lines using the Java Modeling Language [37].

Software Model Checking. In software model checking, the program is translated into a graph of states and transitions [14]. The analysis of such a graph is the verification process of model checking. Because model checkers are able to handle different values of variables, they can be used to simulate different feature selections. Thus, model checkers can be efficiently used for family-based verification of a metaproduct. There are also approaches and tools beyond variability encoding, but they analyze models rather than source code [15, 68] (e.g., product-lines transition systems [20]), and thus are out of our scope.

The Software Variant Generation System (SVGS) [51] uses variability encoding to verify annotation-based product lines, such as the Linux kernel. FeatureIDE [69, 71] provides support for family-based model checking with the core implementation of the

	Annotation-Based	Composition-Based	Independent
Sampling	MoSo-PoLiTe [46], Undertaker [66]		GeneticTestCaseGeneration [18], Pacogen [23], Perrouin et al. [48], PLEDGE [22], SPLCATool [25]
Testing	cse [60], shared-execution [34], splmonitor [33], vai [30]	GATE [19], Kesit [73], SharQ [74], Varex [43]	Asadal [35], MATE [54], ParTeG [75], ScenTED-DF [64], SPLTester [32]
Type Checking	CIDE [27], SVGS [51], TypeChef [29]	DeltaJ [58], FeatureTweezer [6], Fuji [5], SafeComp [9]	
Static Analysis	Emergo [56], SPL^{LIFT} [12]	ACVTool [36], SpeK [59]	
Model Checking	SVGS [51]	FeatureIDE [69], jpf-bdd [30], SPLverifier [7]	
Theorem Proving		FeatureIDE [69]	
Consistency Checking	FeatureIDE [69], LIFE [63], Undertaker [67]	FeatureIDE [69]	
Non-Functional Properties		FBPM [62]	ClaferMOO [1], SPLConqueror [61]
Code Metrics	CIDE [27], Colligens [41], cppstats [38], FeatureIDE [69]	AJDTStats [28], AJStats [2], FeatureIDE [69]	

Table 1: Analysis tools with respect to implementation technique and software analysis.

JavaPathfinder [21] and supports runtime assertion checking with the Java Modeling Language [37]. Another metaproduct that can be verified efficiently with the JavaPathfinder extension JPF-BDD [30] can be generated with FeatureHouse [4]. The tool SPLverifier [7] supports family-based model checking with JavaPathfinder for Java or the CPAchecker [11] for C, using aspects to weave specifications into feature-oriented product-lines.

Theorem Proving. The verification technique theorem proving is a deductive approach to prove logical formulas. First, the program and its specification (e.g., a contract that specifies the behavior of each method) are translated into logical formulas (a.k.a. proof obligations). Then the formulas are used to proof correctness of the program.

Next to the metaproduct for model checking, FeatureIDE [69, 71] also provides a metaproduct with variability-aware contracts of the Java Modeling Language [37], for family-based theorem proving of feature-oriented product-lines. Other variability-aware tool support, or support for other efficient techniques to verify software product lines using theorem proving (e.g., proof composition [72]), is currently missing.

Consistency Checking. The variability used in implementation artifacts (e.g., features in `#ifdef` statements) needs to be valid according to variability defined at the feature model. Consistency checking analyses the usage of features; whether a feature has a corresponding implementation and vice versa [70, 67]. Additionally, consistency checking analyzes the feature dependencies with the usage at the source code (e.g., dead or superfluous code in case of incorrect combination of `#ifdef` statements) [66].

The tool Undertaker [67] can detect dead and superfluous code in preprocessor annotated C programs, and was initially designed to analyze the Linux kernel. The Linux Feature Explorer (LIFE) [63] provides further consistency checks of unused and undefined features for the Linux kernel using Undertaker [67] and Kconfig. The tool FeatureIDE [69] provides consistency checking for preprocessors, aspect-oriented, feature-oriented, and delta-oriented programming, such as unused or undefined features, tautologies or contradictions in preprocessor annotations, and dead code analysis.

5. FURTHER PRODUCT-LINE ANALYSES

In this section, we discuss tools for analyses of product lines that are beyond verification and testing. We present product-line tools for retrieving non-functional properties and code metrics.

Non-Functional Properties. A goal of software development is to optimize non-functional properties, such as footprint, performance, and energy consumption [57]. In single-system engineering, such properties can be reached by a specialized implementation for the properties defined by stakeholders. However, it is even more challenging to automatically determine the optimal product for a given product line related to a given non-functional property (e.g., with the lowest energy consumption). The goal of research on non-functional properties is to predict such properties for all products based on measurements of some products.

The tool SPLConqueror [61] is a framework to measure and optimize non-functional properties in software product lines. It supports user-defined measurements, and automatic computation of optimized products. Siegmund et al. provide an extension of FeatureHouse [4] for family-based performance measurements (FBPM) [62]. The tool ClaferMOO, a standalone extension of the Clafer modeling language [1], provides multi-objective optimization for attributed feature models with quality attributes and optimization objectives.

Code Metrics. Code metrics are used to compare analyses results and to evaluate their expressiveness. In software product-line analyses often metrics such as lines of code and numbers of features are used. However, such metrics do not take feature dependencies, and variability in the source code into account. To compare analysis results for different software product lines, other metrics are required.

The tool cppstats [38] is a C-program analyzer, which can express variability of programs with preprocessor annotations beyond quantity of annotations. The tool Colligens [41] provides information about preprocessor directives for C. CIDE [27] can collect several statistics about the source code, annotations and interactions between annotations for virtual separation of concerns. The tool AJStats [2] collects statistics about AspectJ programs. It collects amount and lines of code of classes, interfaces, their methods, constructors and field. Furthermore it collects this statistics for aspects, their pointcuts, advices and inter-type declarations. AJDTStats [28] collect statistics how aspects are used in AspectJ programs, such as shared joint points,

	Product-Based	Family-Based	Family-Product-Based	Feature-Based
Sampling	GeneticTestCaseGeneration [18], MoSo-PoLiTe [46], Pacogen [23], Perrouin et al. [48], PLEDGE [22], SPLCATool [25], Undertaker [66]			
Testing	Asadal [35], GATE [19], Kesit [73], MATE [54], ParTeG [75], ScenTED-DF [64], SharQ [74], SPLTester [32]	shared-execution [34], vai [30], Varex [43]	cse [60], splmonitor [33]	
Type Checking	Fuji [5]	CIDE [27], DeltaJ [58], FeatureTweezer [6], Fuji [5], SafeComp [9], SVGS [51], TypeChef [29]		Fuji [5]
Static Analysis	ACVTool [36], SpeK [59]	Emergo [56], SPL^{LIFT} [12]		
Model Checking		FeatureIDE [69], jpf-bdd [30], SPLverifier [7], SVGS [51]		
Theorem Proving		FeatureIDE [69]		
Consistency Checking		FeatureIDE [69], LIFE [63], Undertaker [67]		
Non-Functional Properties		FBPM [62]	ClaferMOO [1], SPLConqueror [61]	
Code Metrics		CIDE [27], Colligens [41], cppstats [38], FeatureIDE [69]		AJDTStats [28], AJStats [2], FeatureIDE [69]

Table 2: Analysis tools with respect to analysis strategy and software analysis. (Invalid combinations are marked with gray)

and homogeneous extension. FeatureIDE [69] provides statistics about the software product line, usage of preprocessor directives, and about the implementation with feature-oriented programming.

6. OVERVIEW ON TOOL SUPPORT

In the previous sections, we gave an overview on analysis tools for software product lines. In Table 1, we summarize all these tools and categorize them into the analysis technique they implement and into the product-line implementation technique they are specialized for. For each pair of composition mechanism and analysis technique there is at least one tool that can be used or adopted, except for theorem proving. Because sampling and non-functional properties are generally independent of the generation mechanism, there are not many tools that are specialized on one mechanism. In contrast, the other analysis techniques often depend on the generation mechanism, because they have to handle the variability defined in the source code. The techniques sampling (7 tools), testing (13 tools), type checking (7 tools), and code metrics (7 tools) are well supported for both, annotation-based and composition-based product-lines with at least three tools for either of them. In contrast, static analysis (4 tools), model checking (4 tools), theorem proving (1 tool), consistency checking (3 tools), and non-functional properties (3 tools) are currently less good supported. Especially for theorem proving of product lines, there is currently only one implementation available.

Some tools are specialized on one specific generation tool (e.g., AHEAD [9]) or a specific programming language (e.g., C). However, a tool can be reused and adapted to support product lines with an equivalent generation technique. For example, tools for virtual separation of concerns should be easy to extend to support preprocessors, and vice versa. Furthermore, tools that operate on metaproducts can be applied to other metaproducts of other implementation techniques. Also tools specialized for one program generator (e.g., FeatureHouse [4]) can be used to analyze product lines of another generator (e.g., AHEAD [9]), if the generation mechanism is similar and a translation from one to another generator is possible [69].

Tools for the same analysis technique can be developed with different analysis strategies. In Table 2, we categorize the tools into the analysis strategies they use to scale an analysis technique from single-system engineering to product lines. Because family-based analysis is an efficient way to analyze the product line [68], most implementations are family-based (21 tools) for all analyses techniques, except for testing and sampling. The majority of tools for testing and sampling are product-based. For sampling it is obvious, because sampling is a product-based approach itself, so there cannot be any tool for other strategies. Testing is usually done on a generated product, so the most tools for testing are product-based (8 tools) or family-product-based (2 tools). Family-based testing is only developed recently in the last 2 years (3 tools). Except of Fuji [5] and code metrics, tools for feature-based analyses are currently missing, because current approaches need knowledge about the whole product line or at least about a specific product. To reason about features individually can be very efficient, because feature combinations do not need to be considered [68]. However, only incomplete programs can be analyzed, which is hard and can only lead to limited results, what might be one reason that feature-based analysis tools are rare.

Tools operate on different kinds of representations of the product-line. There are three types of tool strategies; they can analyze products, use a variability encoding, or are variability aware and consider the variability of the product line. In Table 3, we categorize the tools into these strategies with respect to their analyses technique. A main goal of current research in product-line analysis is to apply analyses to the product line while considering its variability. Many variability-aware tools where developed that can operate on the product line as-is (24 tools). However, static analyses, model checking, and theorem proving do still need a variability-encoding of the product line, because they reuse an analysis tool for single systems as-is. To analyze a product-line without variability encoding, new tools need to be developed that are aware of variability.

	Product-Based	Variability-Aware	Variability Encoding
Sampling	GeneticTestCaseGeneration [18], MoSo-PoLiTe [46], Pacogen [23], Perrouin et al. [48], PLEDGE [22], SPLCATool [25], Undertaker [66]		
Testing	Asadal [35], cse [60], GATE [19], Kesit [73], MATE [54], ParTeG [75], SharQ [74], ScenTED-DF [64], splmonitor [33], SPLTester [32]	shared-execution [34], vai [30], Varex [43]	
Type Checking	Fuji [5]	CIDE [27], DeltaJ [58], FeatureTweezer [6], Fuji [5], SafeComp [9], SVGS [51], TypeChef [29]	
Static Analysis	ACVTool [36], SpeK [59]	Emergo [56], SPL^{LIFT} [12]	
Model Checking			FeatureIDE [69], jpf-bdd [30], SPLverifier [7], SVGS [51]
Theorem Proving			FeatureIDE [69]
Consistency Checking		FeatureIDE [69], LIFE [63], Undertaker [67]	
Non-Functional Properties		ClaferMOO [1], FBPM [62], SPLConqueror [61]	
Code Metrics		AJDTStats [28], AJStats [2], CIDE [27], Colligens [41], cppstats [38], FeatureIDE [69]	

Table 3: Analysis tools with respect to strategy of the tool and software analysis. (Invalid combinations are marked with gray)

7. CONCLUSION

Efficient analysis of software is essential for its development. This holds especially for software product lines, because an error may only occur in certain products. For development of software product lines, tools that implement software analyses are necessary. An overview on such tools helps researchers, lecturers, students, and practitioners to decide whether a tool for a certain analysis strategy or generation technique exists that could be used (e.g., for development of commercial-quality tools, or proof of concepts). Furthermore, the effort for development of new tools can be reduced through knowledge about existing tools. Based on a recent survey [68], we give an overview on such tools. To access these tools easily, we provide a website that collects them, and provides links directly to the tools websites. Analysis is only one part of product-line engineering, thus we have also collected tools for product generation, refactoring, and development environments for product lines. Furthermore, we provide additional information about the tools, such as the supported generation mechanism (e.g., preprocessors), the supported programming languages (e.g., Java), and related tools (e.g., to find an IDE that works in combination). Tools for product configuration and domain modeling are currently not covered but may be in future.

8. ACKNOWLEDGMENTS

We thank Wolfram Fenske for his help on refactoring tools for software product lines, which are included on our website. We also thank Norbert Siegmund for help on tools for non-functional properties. This work is partially funded by DFG grant SA 465/34-2.

9. REFERENCES

[1] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia Hui Jimmy Liang, and Krzysztof Czarnecki. Clafer Tools for Product Line Engineering. In *SPLC*, pages 130–135. ACM, 2013.

[2] Sven Apel and Don Batory. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *JOT*, 9(1):117–142, 2010.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[4] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[5] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access Control in Feature-Oriented Programming. *SCP*, 77(3):174–187, 2012.

[6] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-Independent Reference Checking in Software Product Lines. In *FOSD*, pages 65–71. ACM, 2010.

[7] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions Using Feature-Aware Verification. In *ASE*, pages 372–375. IEEE, 2011.

[8] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pages 482–491. IEEE, 2013.

[9] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *GTTSE*, pages 3–35. Springer, 2006.

[10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, pages 184–190. Springer, 2011.

[12] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI*, pages 355–364. ACM, 2013.

[13] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *IST*, 2014. To appear.

[14] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[15] Andreas Classen, Patrick Heymans, Pierre-Yves

Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *ICSE*, pages 335–344. ACM, 2010.

[16] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[17] Paulo Anselmo Da Mota Silveira Neto, Ivan Do Carmo Machado, John D. McGregor, Eduardo Santana De Almeida, and Silvio Romero De Lemos Meira. A Systematic Mapping Study of Software Product Lines Testing. *IST*, 53(5):407–423, 2011.

[18] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *CAiSE*, volume 7328 of *Lecture Notes in Computer Science*, pages 613–628. Springer, 2012.

[19] Yankui Feng, Xiaodong Liu, and Jon Kerridge. A Product Line Based Aspect-Oriented Generative Unit Testing Approach to Building Quality Components. In *COMPSAC*, volume 2, pages 403–408. IEEE, 2007.

[20] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS*, pages 113–131. Springer, 2008.

[21] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[22] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. PLEDGE: A Product Line Editor and Test Generation Tool. In *SPLC*, pages 126–129. ACM, 2013.

[23] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen: Automatic Generation of Pairwise Test Configurations From Feature Models. In *ISSRE*, pages 120–129. IEEE, 2011.

[24] Praveen Jayaraman, Jon Whittle, Ahmed M. Elkhodary, and Hassan Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MODELS*, pages 151–165. Springer, 2007.

[25] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *SPLC*, pages 46–55. ACM, 2012.

[26] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[27] Christian Kästner and Sven Apel. Virtual Separation of Concerns-a Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[28] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

[29] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*, pages 805–824. ACM, 2011.

[30] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In *FOSD*, pages 1–8. ACM, 2012.

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242. Springer, 1997.

[32] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *AOSD*, pages 57—68. ACM, 2011.

[33] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *RV*, pages 285–299. Springer, 2010.

[34] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared Execution for Efficiently Testing Product Lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[35] Kyungseok Kim, Hyejung Kim, Miyoung Ahn, Minseok Seo, Yeop Chang, and Kyo C Kang. ASADAL: a Tool System for Co-Development of Software and Test Environment Based on Product Line Engineering. In *ICSE*, pages 783–786. ACM, 2006.

[36] Herbert Klaeren, Elke Pulvermueller, Awais Rashid, and Andreas Speck. Aspect Composition Applying the Design by Contract Principle. In *GCSE*, pages 57–69. Springer, 2001.

[37] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[38] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of c Code. In *AOSD*, pages 191–202. ACM, 2011.

[39] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *ESECFSE*, pages 81–91. ACM, August 2013.

[40] Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. A Systematic Review of Domain Analysis Tools. *Information and Software Technology*, 52(1):1–13, 2010.

[41] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. Colligens: A tool to support the development of preprocessor-based software product lines in c. In *CBSoft*, 2013.

[42] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.

[43] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *ICSE*. ACM, 2014. To appear.

[44] Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *CSUR*, 43(2):11:1–11:29, 2011.

[45] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2010.

[46] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: tool support for pairwise

and model-based software product line testing. In *VaMoS*, pages 79–82. ACM, 2011.

[47] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *SPLC*, pages 91–100. ACM, 2013.

[48] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *ICST*, pages 459–468. IEEE, 2010.

[49] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, USA, 2002.

[50] Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *SCP*, 41(1):53–84, 2001.

[51] Hendrik Post and Carsten Sinz. Configuration Lifting: Software Verification meets Software Configuration. In *ASE*, pages 347–350. IEEE, 2008.

[52] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443. Springer, 1997.

[53] pure::systems. pure::variants. Website. Available online at `http://www.pure-systems.com/pure_variants.49.0.html`; visited on June 16th, 2014.

[54] Georg Püschel, Ronny Seiger, and Thomas Schlegel. Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. In *MODIQUITOUS*, 2012.

[55] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *SIGPLAN-SIGACT*, pages 49–61. ACM, 1995.

[56] Márcio Ribeiro, Társis Tolêdo, Johnni Winther, Claus Brabrand, and Paulo Borba. Emergo: A Tool for Improving Maintainability of Preprocessor-Based Product Lines. In *AOSD*, pages 23–26. ACM, 2012.

[57] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Pearson Education, 2012.

[58] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *SPLC*, pages 77–91. Springer, 2010.

[59] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *FOSD*, pages 7:1–7:8. ACM, 2011.

[60] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *FASE*, pages 270–284. Springer, 2012.

[61] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *SQJ*, 20(3-4):487–517, 2012.

[62] Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-Based Performance Measurement. In *GPCE*, pages 95–104. ACM, 2013.

[63] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In *EuroSys*, 2010.

[64] Vanessa Stricker, Andreas Metzger, and Klaus Pohl. Avoiding Redundant Testing in Application Engineering. In *SPLC*, pages 226–240. Springer, 2010.

[65] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, January 2012.

[66] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang 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*, pages 47–60. ACM, 2011.

[67] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *FOSD*, pages 81–86. ACM, 2009.

[68] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 2014. To appear.

[69] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[70] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pages 191–200. IEEE, 2011.

[71] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential synergies of theorem proving and model checking for software product lines. In *SPLC*. ACM, 2014. To appear.

[72] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *VAST*, pages 270–277. IEEE, 2011.

[73] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing Software Product Lines Using Incremental Test Generation. In *ISSRE*, pages 249–258. IEEE, 2008.

[74] Engin Uzuncaova, Sarfraz Khurshid, and Don Batory. Incremental Test Generation for Software Product Lines. *TSE*, 36(3):309–322, 2010.

[75] Stephan Weißleder, Dehla Sokenou, and Bernd-Holger Schlingloff. Reusing State Machines for Automatic Test Generation in Product Lines. *MoTiP*, pages 19–28, 2008.