A Concise Classification of Reverse Engineering Approaches for Software Product Lines

Rehman Arshad,Kung-Kiu-Lau
School of Computer Science, University of Manchester
Kilburn House, Oxford Road, Manchester, United Kingdom
e-mail: rehman.arshad, kung-kiu.lau @manchester.ac.uk

***Abstract*—Reverse engineering in product lines means identification of feature locations in the source code or formation of the non-redundant feature model from descriptive documents. The feature identification can be represented by feature to code trace, graphical notations or tools based view. For adopting a specific approach, it is very important to know how it works, the kind of expertise needed to use it, the kind of tool support that is there, the format of the required input for using that approach, the output notation that it can provide, the related shortcomings that cannot be avoided and the kind of pre-requisite work each approach demands. Based on these parameters, this paper provides a classification of the reverse engineering approaches related to software product lines. Such classification can help the product line engineers or relevant researchers to narrow down the practical options for their implementation and to obtain the better understanding of reverse engineering in product lines.**

Keywords: Product Line Engineering; Reverse Engineering; Static Analysis; Dynamic Analysis; Textual Analysis; Hybrid Analysis; Feature Location.

I. INTRODUCTION

”The output of a reverse engineering activity is synthesized, higher-level information that enables the reverse engineer to better reason about the system and to evolve it in an effective manner” [1]. Usually, the result of reverse engineering is in higher notation of abstraction in order to understand the system. Output can be a model, graphical chart, re-structured code or some notation that can express the system in a feasible way.

The idea behind a software product line is to make different customized software products by using same platform that can support different variations in all the products. The process of constructing and managing such common platform (product line) is known as product line engineering [2].

A product line is usually composed of features ranging from few dozens to several hundreds [2]. These features are related to each other by well-defined constraints [3]. Without very extensive documentation and trace, it is almost impossible for product line engineers to understand the composition of code in terms of features. The process of reverse engineering in product lines is used for finding the feature locations in the code or for constructing a non-redundant feature model from descriptive documentation. With the evolution of the product line, the major purpose of reverse engineering is to keep code and variability modelling synchronised and understandable.

One of the concerns for product line practitioners and programmers is to know the difference in applicability of different reverse engineering approaches according to domain, available in-house expertise, tool support and required notation of extraction after reverse engineering. This short survey provides a basic classification for software product line engineers and programmers to know the difference between reverse engineering approaches for product lines, the tools that come with some approaches, kind of output provided by each approach, kind of input needed by each approach, associated shortcomings to each approach, prerequisites for implementing an approach and kind of expertise needed in order to implement an approach. Such comparison between these approaches will help in the selection of an approach over others on the basis of compatibility with all such parameters. This classification can also help novices to understand what it takes to do reverse engineering for software product lines.

The term technique and approach should not be confused in this survey. One approach can use various well-defined techniques for its implementation where as an approach is the way in which a process uses many techniques in order to get the results, e.g., *Language Independent Approach* [4] is an approach of reverse engineering that is based on techniques of Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI). Similarly, *Static Analysis* is a kind of analysis technique in reverse engineering and this technique can be used by multiple software product line reverse engineering approaches. The presented classification in this paper can help the product line engineers and relevant researchers to narrow down the practical options for their implementation instead of wasting time on comparing all such options.

The remainder of this paper is organised as follows. Section 2 explains the related work. Section 3 includes the framework of classification. Section 4 includes the classified approaches based on the framework. Section 5 provides available tool support for each approach and major shortcomings of approaches. Section 6 is the final section that includes conclusion.

II. RELATED WORK

There are hundreds of approaches in the reverse engineering but most of them are not applicable in the domain of product lines. In this paper only the approaches that meet the following criteria have been selected: they are for product lines in particular, related to product variants, relevant to feature identification/formation in the complex system families and tackle the variability of the product variants.
Therefore, many well-known general reverse engineering techniques like LSI [5], Probabilistic Latent Semantic Indexing (PBLSI) [6] and NL-Queries [7] are not part of this paper. These techniques can be part of complex reverse engineering approaches related to software product lines but as standalone techniques they are not relevant. This is because a product line is constructed in terms of features and their variations, and general techniques cannot produce results in terms of features and variants. It is not the intention of this paper to include the approaches that use (*reuse*) artefacts to construct a product line. Approaches with the sole purpose of reverse engineering are considered only, therefore approaches like *clone and own* are not part of this classification because they are mainly used to *reuse* not to *reverse engineer*.

This paper covers more than thirty approaches of reverse engineering in the field of software product lines. Each approach uses many techniques of reverse engineering. Techniques like LSI, FCA, etc., do not belong to a specific domain. The way an approach uses these techniques determines whether the approach is suitable for product lines or not. Few surveys have classified reverse engineering techniques but none have done it solely for software product lines and their angle of interest is quite different, e.g., Bogdan Dit's survey [8] is the closest one because it covers the identification of feature locations. Work of Michael L. Nelson [9] covers automation of reverse engineering in legacy system. Purpose of M. Spiros and K. Moshe's survey [10] is to classify the tool support for specific operating systems. Such classifications and surveys cannot help in the domain of software product lines. They cannot help in deciding the applicability of reverse engineering because they do not discuss the classification parameters with respect to software product lines. A classified tool of reverse engineering may be great from operating system's point of view but can be useless for software product lines at the same time. Overall, no such survey exists at the moment that has discussed the reverse engineering from software product line's point of view.

III. Framework of Classification

Table I. shows the framework of classification in terms of different parameters. The parameters used for classification are;

- Analysis Technique
- Required input notation
- Generated output notation
- Phase Compatibility
- Required Expertise
- Pre-Requisite Implementation

These parameters are selected by considering their importance for applicability of practical implementation. Analysis techniques define the type of analysis used by an approach for reverse engineering. Required input notation classifies the approaches based on the input they require for execution. Generated output notation classifies the approach based on the type of output produced by each approach. Phase compatibility means whether an approach is suitable for construction or maintenance of a product line. Required expertise groups the approaches based on the kind of techniques they use and pre-requisite implementation classifies the approaches based on the kind of work they require before implementation. Further classification of these parameters is presented in Table I.

TABLE I. REVERSE ENGINEERING: FRAMEWORK OF CLASSIFICATION

Parameter of Classification	**Classification**	
Analysis Technique	Static	
	Textual	
	Dynamic	
	Hybrid	
Req. Input Notation	Source Code	
	Feature Set	
	Description	
Output Notation	Feature Model	
	Code	
	View Based	Concept Lattices and Graphs
		Feature to Code Trace
Phase Compatibility	Construction	
	Maintenance	
Required Expertise	Profiling	
	FCA	
	LSI	
	Vector Space Modelling (VSM)	
	Domain Knowledge	
	Natural Language Processing (NLP)	
Pre-Requisites	Profiling (Code Instrumentation)	
	Approach Centric	

IV. Classification of Reverse Engineering Approaches for Software Product Lines

This section will classify the reverse engineering approaches for software product lines based on analysis technique, input notation required by each approach, output notation, phase compatibility, pre-requisite implementation required by each approach and expertise required by each approach. All these classification parameters are presented in the following sections.

A. Classification based on Analysis Technique

Analysis techniques in reverse engineering are classified as follows: [8]

- Static Analysis Techniques
- Textual Analysis Techniques
- Dynamic Analysis Techniques
- Hybrid Analysis Techniques

1) Static Analysis Techniques: Static feature location techniques are based on structural information of the code. They consider control flow, data flow and dependencies in the code to identify features. These techniques work by building a model of states of the program and then determine all possible routes of the program at each step. To design such approach one has to keep the balance between preciseness and granularity and some abstraction is used to consider which steps should be added in the static analysis model [11].

These techniques are based on the control structure of the source code, hence their result has very good recall but the major drawback is lack of precision. False positive results are very common in static techniques as these techniques work on user-defined model of control flow rather than the actual trace of the program. The biggest advantage is the future re-usability.

The output of such techniques can be a configuration matrix, a dependency graph or re-formation of the actual source code. Usually, these techniques are used to extract a dependency matrix between the source code and features in order to understand the relation between code and the variability model composed by features. RecoVar [12], Language Independent Approach [4], Dependency Graph [13], Concern Graph [14], Automatic Generation [15], Concern Identification [16] and Semi-Automatic Approach [17] are some of the approaches related to the product line engineering based on static analysis.

2) Textual Analysis Techniques: Few researchers [18] referred textual as a static technique but it is quite different from a standard static technique. Textual analysis does not need any abstraction model and uses the query-based input to match the words with identifiers and comments in the code.

Most textual analysis reverse engineering techniques produce feature locations as an output. These code locations are displayed either by concept lattices (if Formal Concept Analysis is used) or by dependency graphs. Examples of such approaches are Combining

FCA with IR [18] and Source Code Retrieval [19]. Few approaches produce feature models from the provided description or the feature-set as description. Examples of these approaches include Evolutionary Algorithms [20], Reverse Engineering Feature Models [40] and Feature Models from Software Configurations [21]. Few textual based approaches also extract and show the code in terms of variability, e.g., Product Variants [22] and few represent domain concepts after extracting them from the code in order to provide understanding of the code in simple domain terms, e.g., Natural Language Parsing [23].

The biggest problem is the user designed queries that are responsible for almost the whole analysis and quality or accuracy of the output. Another problem is *polysemy* and implicit implementation of the feature across many locations.

3) Dynamic Analysis Techniques: Dynamic analysis uses execution trace of the program to follow and identify the feature locations by following running code. Test scenarios or profiling is needed in order to design an execution trace with respect to some feature. Profiling is instrumentation of the code and it is a difficult task. Usually one scenario can only involve one feature, hence in case of hundreds of features, dynamic analysis becomes more complex. For every new profiling, old results are useless whereas in static we can reuse the rules of abstraction as many times as we want with continuous refinement.

Dynamic analysis output is always a trace that shows feature locations in the code. This relation is represented either as concept lattices or view-based tools. The abstraction level of code in the trace varies from approach to approach. Dynamic Feature Traces [24], Feature to code trace [25], Focused views on Execution Traces [26], Software evolution Analysis [27], Trace Dependency Analysis [28], Featureous [29], Embedded Call-Graphs [30], Scenario-Driven Dynamic Analysis [31] and Concept Analysis [32] are examples of product line approaches based on dynamic analysis.

4) Hybrid Analysis Techniques: A hybrid analysis in reverse engineering can be a combination of Dynamic-Static, Dynamic-Textual, Textual-static or Dynamic-Textual-static analysis. Hybrid analysis can join *recall* of static and *precision* of dynamic analysis. *Recall* is required in order to make dynamic analysis reusable in the future. So a static analysis can obviate the collection of certain information and dynamic can run over that collection in order to get better results. Also, many approaches like [33] use one analysis technique just to rank the elements of the code so this ranking of feature relevancy will be considered in the final results in order to increase accuracy.

Hybrid techniques provide feature locations either by using concept lattices or graphs. Static and Dynamic Analysis [34], Cerberus [33], Sniafl [35], Locating Features in Source Code [36], Using Landmarks and Barriers [37] and A Heuristic-Based Approach [38] are examples of reverse engineering approaches based on hybrid analysis.

Few approaches that cannot be fit in the classification are the ones that are dependent on pure data mining in order to correlate product variants to dependency graphs in order to predict the influence of one feature on others, e.g., [39]. The selection of analysis technique is based on many parameters like availability of the abstraction model, trade-off between false positive and accuracy, availability of profiling to run every feature and most importantly the kind of reverse engineering needed. The whole classification of this section is summarised in Table II.

TABLE II. REVERSE ENGINEERING:CLASSIFICATION OF REVERSE ENGINEERING APPROACHES BASED ON ANALYSIS

Reverse Engineering Approaches	**Analysis Classification**
RECoVar [12], Language Independent Approach [4], Dependence Graph [13], Concern Graphs [14], Concern Identification [16], Automatic Generation [15], Semi-Automatic Approach for Extraction [17]	Static
Product Variants [22], Natural language Parsing [23], Evolutionary Algorithms [20], Software Configurations using FCA [21], Source Code Retrieval [19], Combining FCA with IR [18], Reverse Engineering Feature Models [40]	Textual
Dynamic Feature Traces [24], STRADA [25], Call-Graphs [30], Focused views on Execution Traces [26], Concept Analysis [32], Trace Dependency Analysis [28], Scenario Driven Dynamic Analysis [31], Featureous [29], Software Evolution Analysis [27]	Dynamic
Static and Dynamic Analysis [34], Cerberus [33], Heuristic-Based Approach [38], Landmarks and Barriers [37], Locating Features in Source Code [36], SNIAFL [35]	Hybrid

B. Classification based on Input and Output

After selecting an appropriate analysis technique on grounds of compatibility and associated shortcomings, it is very important to know about required input notation and generated output notation of each approach. Some input notations are not compatible with some product lines implemented form and a lot of work is needed in order to transform code into specific input notation. To avoid extra work, one can select an approach that is most appropriate for the environment. The required input notation can be classified as *Source Code*, *Feature Set* and *Description Based Input*. *Feature Set* means configuration matrix or product-feature mapping in some notation where *Description* includes user Queries, Document-Corpus and Textual Input like natural language text etc.,

Similarly, output can also be classified into *Feature Model*, *Generated Code* and *View Based Output*. View Based Output can further be classified into concept Lattices or graphical notations and ranked Based Mapping or Feature to Code Trace.

Few approaches produce feature models as output [20] [21] [40]. Few transform code into core and variability parts [4] [22]. Few approaches generate feature-code trace [15] [17] [19] [23]- [25] [28] [31]- [36] [38]. Few generate output in the form of concept lattices or graphs [12]- [14] [16] [18] [27] [30] [37]. Concept lattices are different from general graphs because they are generated by defining the FCA and can be manipulated by changing the formal contexts whereas general graphs usually show variability models extracted from the code.

Hybrid approaches in this category use one analysis technique to reinforce the results and then use another technique on the generated output of the first one. Such hybrid approaches show results in the form of ranked based mapping where each mapping has a value based on its validity. Ranked based mapping is also a trace but it includes the ranking of the traces. Few approaches like [26] [29] generate both

TABLE III. REVERSE ENGINEERING: CLASSIFICATION BASED ON REQUIRED INPUT

Required Input	**Reverse Engineering Techniques**
Source Code	RECoVar [12], Call-Graphs [30], Concern Identification [16], Scenario Driven Dynamic Analysis [31], Featureous [29], Language Independent Approach [4], Semi-Automatic Approach for Extraction [17], SNIAFL [35], Static and Dynamic Analysis [34], Cerberus [33], Bug Localization [19], Focused views on Execution Traces [26], Heuristic-Based Approach [38], Dependence Graph [13], Software Evolution Analysis [27], Concern Graphs [14], Concept Analysis [32]
Feature Set	Product Variants [22], Natural language Parsing [23], Dynamic Feature Traces [24], Evolutionary Algorithms [20], Software Configurations using FCA [21], Static and Dynamic Analysis [34], STRADA [25]
Description (Queries, Document-Corpus, Textual input)	Cerberus [33], Landmarks and Barriers [37], Locating Features in Source Code [36], Source Code Retrieval [19], Trace Dependency Analysis [28], SNIAFL [35] Combining FCA with IR [18], Automatic Generation [15], Reverse Engineering Feature Models [40]

TABLE IV. REVERSE ENGINEERING: CLASSIFICATION BASED ON GENERATED OUTPUT

Generated Output		**Reverse Engineering Techniques**
Feature Model		Evolutionary Algorithms [20], Software Configurations using FCA [21], Reverse Engineering Feature Models [40]
Code		Product Variants [22], Language Independent Approach [4]
View-Based	Concept Lattices or Graphical notations	Combining FCA with IR [18], Landmarks and Barriers [37], Call-Graphs [30], Concern Identification [16], Dependence Graph [13], Concern Graphs [14], Software Evolution Analysis [27], RECoVar [12], Focused views on Execution Traces [26], Featureous [29]
	Ranked Based Mapping or Feature to Code Trace	Cerberus [33], SNIAFL [35], Source Code Retrieval [19], Scenario Driven Dynamic Analysis [31], STRADA [25], Natural language Parsing [23], Trace Dependency Analysis [28], Concept Analysis [32], Dynamic Feature Traces [24], Static and Dynamic Analysis [34], Locating Features in Source Code [36], Heuristic-Based Approach [38], Semi-Automatic Approach for Extraction [17], Focused views on Execution Traces [26], Featureous [29], Automatic Generation [15]

trace and graphical views. Table III. and IV. show the classification based on these parameters.

C. Classification based on Phase Compatibility, Pre-requisite Implementation and Required Expertise

Table V. shows the pre-requisites for implementing an approach. Pre-requisites have classified into approach centric process, i.e., *macro constant's selection, landmarks method selection, domain concepts, corpus extraction* and *profiling*. Profiling is the most common pre-requisite. RECoVar [12] is an approach that requires selection of the macro constants before it can be applied. It shows code based variability by extracting a model from the code. Users have to define the macro constants in the code to use them in conditional compiling while generating the model. Such macro constants can be *if-def* blocks or anything that can define a variation in pre-compilation and they are called variation points. Another approach Landmarks and Barriers [37] demands selection of *landmark* methods. Landmark methods are those that have a key role in execution of a feature. Hence, in order to select landmark features one must have to know that feature composition in terms of code. *Barrier* methods are those methods that do not have major importance from a feature point of view and they have to be selected in order to decrease the size of generated variability graph. Combining FCA with IR [18] demands generation of the document corpus by LSI. Document corpus is the generation of the part of the code that matches the user queries and it should be in vector space form which is a well known form in LSI. FCA uses this notation to start matching and producing the output in the form of concept lattices. Dependence Graph [13] needs identification and selection of the nodes that should be included in the search graph in order to search the implementation of a feature. The relevant code parts cannot be selected unless one has the knowledge and some familiarity with the domain and composition of the features in terms of code. So some code understanding and domain knowledge is must before executing this approach. In case of Reverse Engineering Feature Models [40], domain knowledge is needed because domain expert have to select the parent of each feature at each step and correct decisions require code and domain knowledge.

Table VI. shows the phase compatibility classification. *Phase Compatibility* shows whether an approach is suitable to use in the construction of a product line or in the maintenance of a product line. There are several approaches that are not designed for the maintenance or evolution but for the construction of a product line and hence they should be used for this purpose, e.g., approaches that can produce *Feature Models* are more appropriate to use in

 ISBN: 978-1-61208-498-5

TABLE V. REVERSE ENGINEERING: PRE-REQUISITE REQUIREMENTS

Pre-Requisite Implementation	Reverse Engineering Techniques
Profiling	Dynamic Feature Traces [24], Scenario Driven Dynamic Analysis [31] Trace Dependency Analysis [28], Featureous [29], Locating Features in Source Code [36], Static and Dynamic Analysis [34], Cerberus [33], STRADA [25], Call-Graphs [30], Focused views on Execution Traces [26], Concept Analysis [32], Heuristic-Based Approach [38], Software Evolution Analysis [27]
Macro Constants Selection	RECoVar [12]
Selection of Landmark Methods	Landmarks and Barriers [37]
Document corpus extraction for LSI	Combining FCA with IR [18]
Understanding of Domain Concepts	Dependence Graph [13], Reverse Engineering Feature Models [40]

constructing a product line rather than maintaining one because a non-redundant *Feature Model* can be achieved from requirement text or product lines initial product-feature documentation. Evolutionary Algorithm [20], Software Configuration using FCA [21] and Reverse Engineering Feature Models [40] are examples of such approaches.

Table VII. shows required expertise that are grouped as FCA, LSI, NLP, Profiling, VSM and Domain Knowledge. Product Variants [22], Concept Analysis [32], Combining FCA with IR [18] and Locating Features in Source Code [36] require the knowledge of FCA. FCA demands the designing of a formal context in which objects are defined in order to generate the model. Product Variants [22], Cerberus [33], Combining FCA with IR [18] and Heuristic-Based Approach [38] require the knowledge of LSI. LSI is a well known textual technique, mostly used in search engines. Natural language Parsing [23] requires Natural Language Processing which is a well established research domain on its own. Dynamic Feature Traces [24], Scenario Driven Dynamic Analysis [31], Trace Dependency Analysis [28], Featureous [29], Locating Features in Source Code [36], Static and Dynamic Analysis [34], Cerberus [33], STRADA [25], Call-Graphs [30], Focused views on Execution Traces [26], Concept Analysis [32], Heuristic-Based Approach [38] and Software Evolution Analysis [27] require profiling. SNIAFL [35] requires the knowledge of VSM. VSM is a special kind of LSI. Finally, Dependence Graph [13] and Reverse Engineering Feature Models [40] need the domain knowledge and the reasons are as stated in the previous section.

V. Available Tool Support, Language Constraint and Shortcomings

This section explains *Primary Tool*, *Secondary Tool*, *Evaluation Language* and *Major Shortcoming* related to each approach. Primary Tool attribute means tools that are specifically made for the approach where secondary Tool means third party tools that are not designed for the specific approach but help in implementing one. Most of the tools are academic where Reverse Engineering Feature Models [40], Focused views on Execution Traces [26] and Featureous [29] have professional tools. Table VIII. and Table X. show the primary tools and secondary tools availability for each approach.

TABLE VI. REVERSE ENGINEERING: PHASE COMPATIBILITY WITH SOFTWARE PRODUCT LINES

Approaches	Phase Compatibility
RECoVar [12], Dependence Graph [13], Concern Graphs [14], Concern Identification [16], Automatic Generation [15], Natural language Parsing [23], Bug Localization [19], Combining FCA with IR [18], Dynamic Feature Traces [24], STRADA [25], Call-Graphs [30], Focused views on Execution Traces [26], Concept Analysis [32], Trace Dependency Analysis [28], Scenario Driven Dynamic Analysis [31], Featureous [29], Software Evolution Analysis [27], Static and Dynamic Analysis [34], Cerberus [33], Heuristic-Based Approach [38], Landmarks and Barriers [37], Locating Features in Source Code [36], SNIAFL [35]	Maintenance
Product Variants [22], Semi-Automatic Approach for Extraction [17], Evolutionary Algorithms [20], Software Configurations using FCA [21], Language Independent Approach [4], Reverse Engineering Feature Models [40]	Construction

TABLE VII. REVERSE ENGINEERING: REQUIRED EXPERTISE

Approaches	Required Expertise
Product Variants [22], Concept Analysis [32], Combining FCA with IR [18], Locating Features in Source Code [36]	FCA
Product Variants [22], Cerberus [33], Combining FCA with IR [18], Heuristic-Based Approach [38]	LSI
Natural language Parsing [23]	NLP
Dynamic Feature Traces [24], Scenario Driven Dynamic Analysis [31], Trace Dependency Analysis [28], Featureous [29], Locating Features in Source Code [36], Static and Dynamic Analysis [34], Cerberus [33], STRADA [25], Call-Graphs [30], Focused views on Execution Traces [26], Concept Analysis [32], Heuristic-Based Approach [38], Software Evolution Analysis [27]	Profiling
SNIAFL [35]	Vector Space Modelling
Dependence Graph [13], Reverse Engineering Feature Models [40]	Domain Knowledge

Evaluation language shows the language in which an approach has been experimented and validated. Approaches that generate feature models and require description based documents as input are language independent, e.g., Evolutionary Algorithms [20] and Software Configurations using FCA [21]. Few approaches like RECoVar

TABLE VIII. REVERSE ENGINEERING: PRIMARY TOOL SUPPORT

Approach	**Primary Tool**
RECoVar [12], Evolutionary Algorithm [20], Feature Models from Software Configurations [21], CERBERUS [33], Source Code Retrieval [19], Combining FCA with IR [18], Heuristic-Based Approach [38], Dependence Graph [13], SNIAFL [35], Trace Dependency Analysis [28], Locating Features in Source Code [36], Scenario-Driven Dynamic Analysis [31]	NA
Product Variants [22]	Progmodel
Natural Language [23]	Patch Tool
Language Independent [4]	ExtractorPL
Semi Automatic Approach [17]	CIDE
Dynamic Feature Traces [24]	DFT
Static and Dynamic Analysis [34]	Customised BIT
STRADA [25]	STRADA
Focused Views on Execution Traces [26]	CGA-LDX
Concept Analysis [32]	Customised GCC
Software Evolution Analysis [27]	Trace Scrapper
Concern Graphs [14]	FEAT
Automatic Generation [15]	EclipsePlug-in
Concern Identification [16]	CoDEx
Featureous [29]	Featureous
Using Landmarks and Barriers [37]	Prototype Tool
Embedded Call Graphs [30]	Call Graph Prototype
Reverse Engineering Feature Models [40]	CDT TOOLS (LVAT)

[12] are methodologies and hence they can be applied in any language but the approaches like Focused views on Execution Traces [26], Featureous [29] and Call Graph [30] are language dependent as their tools are dependent on the programming language they have designed for. Table IX. shows the evaluation language of each approach.

One major shortcoming is the inability of an approach to consider cross cutting constraints (CTC), e.g., Semi-Automatic Approach for Extraction [17] and Language Independent Approach [4]. Few approaches like Software Configurations using FCA [21] consider CTC but they cannot produce feature model beyond two levels of hierarchy. Results of Dynamic Feature Traces [24], STRADA [25], Source Code Retrieval [19], Concept Analysis [32], Combining FCA with IR [18], Heuristic-Based Approach [38] and Trace Dependency Analysis [28] are highly dependent on the user defined input. This input is approach centric and can be code knowledge, profiling, test scenarios or setting the formal context. More detail is expressed in Table XI. Language constraint, availability of tool and relevant shortcomings are the primary factors to consider one approach over the others.

VI. CONCLUSION

This paper has presented a concise classification of reverse engineering approaches in software product lines. Individual reverse

TABLE IX. REVERSE ENGINEERING: EVALUATION LANGUAGE OF APPROACHES

Approach	**Evaluation Language**
Product Variants [22], RECoVar [12], Evolutionary Algorithm [20], Feature Models from Software Configurations [21]	Language Independent
Natural Language [23], Language Independent [4], Semi Automatic Approach [17], Dynamic Feature Traces [24], Static and Dynamic Analysis [34], CERBERUS [33], STRADA [25], Source Code Retrieval [19], Combining FCA with IR [18], Heuristic-Based Approach [38], Software Evolution Analysis [27], Concern Graph [14], Automatic Generation [15], Concern Identification [16], Featureous [29], Using Landmarks and Barriers [37]	JAVA
Concept Analysis [32], Embedded Call-Graphs [30], Locating Features in Source Code [36], SNIAFL [35], Dependence Graph [13]	C
Source Code Retrieval [19], Focused Views on Execution Traces [26], Reverse Engineering Feature Models [40], Scenario-Driven Dynamic Analysis [31], Embedded Call-Graphs [30], Trace Dependency Analysis [28]	C++

TABLE X. REVERSE ENGINEERING: SECONDARY TOOL SUPPORT

Approach	**Secondary Tool**
Product Variants [22], Natural Language [23], Language Independent [4], Semi Automatic Approach [17], Dynamic Feature Traces [24], STRADA [25], CERBERUS [33], Focused Views on Execution Traces [26], Dependence Graph [13], Software Evolution Analysis [27], Concern Graphs [14], Automatic Generation [15], Locating Features in Source Code [36], Featureous [29], Concern Identification [16], Embedded Call Graphs [30], Using Landmarks and Barriers [37], Reverse Engineering Feature Models [40]	NA
RECoVar [12]	Treeviz, Orange
Feature Models from Software Configurations [21]	FAMA, SPLOT
Source Code Retrieval [19]	Gibbs, LDA++
Combining FCA with IR [18]	SrcML, Columbus
Trace Dependency Analysis [28]	Rational Coverage
Scenario-Driven Dynamic Analysis [31]	JGraph
Evolutionary Algorithm [20]	BETTY
Static and Dynamic Analysis [34]	SA4J
Concept Analysis [32]	Graphlet
Heuristic-Based Approach [38]	MoDeC
SNIAFL [35]	SMART

TABLE XI. REVERSE ENGINEERING: MAJOR SHORTCOMING OF APPROACHES

Approach	Major Shortcoming
Language Independent Approach [4], Semi Automatic Approach [17]	CTC not considered
Dynamic Feature Traces [24], STRADA [25], Source code Retrieval [19], Concept Analysis [32], Trace Dependency Analysis [28], Heuristic-Based Approach [38], Combining FCA with IR [18]	Result dependency on user defined input
RECoVar [12], Reverse Engineering Feature Models using Landmarks and Barriers [37], Dependence Graph [13]	Require code understanding
Natural Language [23], Evolutionary Algorithm [20]	High computation cost
Product Variants [22]	Non-re-usability if feature set changes
Feature Models from Software Configurations [21]	Extract Feature Model for two levels of hierarchy
Static and Dynamic Analysis [34]	Work for only one feature at a time
CERBERUS [33], Locating Features in Source Code [36]	No tool support
Focused Views on Execution Traces [26]	Only work for C/C++ code
Software Evolution Analysis [27], SNIAFL [35], Scenario-Driven Dynamic Analysis [31]	Method implementation neglected
Concern Graphs [14], Concern Identification [16]	Intra-method flow of calls neglected
Automatic Generation [15]	Implicit features neglected
Featureous [29]	JAVA tool dependency
Embedded Call-Graphs [30]	C/C++ tool dependency

engineering techniques that cannot produce results in terms of features and variants of a product line were not considered. The primary aim of this short guide is to present such information that can narrow down the practical options of implementation for the product line engineers so they can discard the non-feasible options of reverse engineering. The reverse engineering in product lines is considered as extraction of artefacts from the code of a product line. However, current approaches do not propose to extract something architectural or in a component notation. Reverse engineering is focused on variability management and features locations at the moment. Future work in this domain can include the approaches that can extract executable architecture from a product line code in order to reuse it across many systems. Hence, the concept of reverse engineering in software product lines should consider the architectural extraction in future.

References

[1] A. C. Telea, "Reverse engineering–recent advances and applications," *Ed. Intech 2012*.

[2] K. Pohl, G. Böckle, and F. Van Der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[3] F. vd Linden, K. Schmid, and E. Rommes, "Software product lines in action: The best industrial practice in product line engineering. secaucus."

[4] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon, "Towards a language-independent approach for reverse-engineering of software product lines," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1064–1071, ACM, 2014.

[5] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[6] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 50–57, ACM, 1999.

[7] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 232–242, IEEE Computer Society, 2009.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[9] M. L. Nelson, "A survey of reverse engineering and program comprehension," *arXiv preprint cs/0503068*, 2005.

[10] J. Raymond, J. Canzanese, O. Matthew, M. Spiros, and K. Moshe, "A survey of reverse engineering tools for the 32-bit microsoft windows environment," *Drexel University*, 2005.

[11] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27, Citeseer, 2003.

[12] B. Zhang and M. Becker, "Recovar: A solution framework towards reverse engineering variability," in *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, pp. 45–48, IEEE, 2013.

[13] K. Chen and V. Rajlich, "Case study of feature location using dependence graph.," in *IWPC*, pp. 241–247, Citeseer, 2000.

[14] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Proceedings of the 24th international conference on Software engineering*, pp. 406–416, ACM, 2002.

[15] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 11–20, ACM, 2005.

[16] M. Trifu, "Improving the dataflow-based concern identification approach," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 109–118, IEEE, 2009.

[17] M. T. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *Software Engineering, IEEE Transactions on*, vol. 38, no. 4, pp. 737–754, 2012.

[18] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pp. 37–48, IEEE, 2007.

[19] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pp. 155–164, IEEE, 2008.

[20] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed, "Reverse engineering feature models with evolutionary algorithms: An exploratory study," in *Search Based Software Engineering*, pp. 168–182, Springer, 2012.

[21] R. Al-Msie'Deen, M. Huchard, A.-D. Seriai, C. Urtado, and S. Vauttier, "Reverse engineering feature models from software configurations using formal concept analysis," in *CLA 2014: Eleventh International Conference on Concept Lattices and Their Applications*, vol. 1252, pp. 95–106, 2014.

[22] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 145–154, IEEE, 2012.

[23] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 156–159, IEEE, 2010.

[24] A. D. Eisenberg and K. De Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 337–346, IEEE, 2005.

[25] A. Egyed, G. Binder, and P. Grunbacher, "Strada: A tool for scenario-based feature-to-code trace detection and analysis," in *Companion to the proceedings of the 29th International Conference on Software Engineering*, pp. 41–42, IEEE Computer Society, 2007.

[26] J. Bohnet, S. Voigt, and J. Dollner, "Locating and understanding features of complex software systems by synchronizing time-, collaboration-and code-focused views on execution traces," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 268–271, IEEE, 2008.

[27] O. Greevy, S. Ducasse, and T. Girba, "Analyzing feature traces to incorporate the semantics of change in software evolution analysis," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 347–356, IEEE, 2005.

[28] A. Egyed, "A scenario-driven approach to trace dependency analysis," *Software Engineering, IEEE Transactions on*, vol. 29, no. 2, pp. 116–132, 2003.

[29] A. Olszak and B. N. Jørgensen, "Featureous: a tool for feature-centric analysis of java software," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 44–45, IEEE, 2010.

[30] J. Bohnet and J. Döllner, "Analyzing feature implementation by visual exploration of architecturally-embedded call-graphs," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pp. 41–48, ACM, 2006.

[31] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Scenario-driven dynamic analysis for comprehending large software systems," pp. 71–80, IEEE, 2006.

[32] T. Eisenbarth, R. Koschke, and D. Simon, "Derivation of feature component maps by means of concept analysis," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pp. 176–179, IEEE, 2001.

[33] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 53–62, IEEE, 2008.

[34] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 236–241, IEEE, 2008.

[35] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 2, pp. 195–226, 2006.

[36] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *Software Engineering, IEEE Transactions on*, vol. 29, no. 3, pp. 210–224, 2003.

[37] N. Walkinshaw, M. Roper, and M. Wood, "Feature location and extraction using landmarks and barriers," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 54–63, IEEE, 2007.

[38] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A heuristic-based approach to identify concepts in execution traces," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 31–40, IEEE, 2010.

[39] B. Zhang and M. Becker, "Reverse engineering complex feature correlations for product line configuration improvement," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pp. 320–327, IEEE, 2014.

[40] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 461–470, IEEE, 2011.

 ISBN: 978-1-61208-498-5