AUTOMATED SOFTWARE ENGINEERING-791: Final Reading

Abstract

This paper will cover the recent development in the field of static security analysis in Product Line Engineering. In this paper I will explore the various works previously detailed and look to explore the ideas they were aimed at solving while exploring how the papers contributed to addressing the previously stated issues. I will then try to generate a thorough "Bigger Picture" of how the various related (and unrelated works) together contribute to the solving of issues and what, if any, new issues have been created by the same for future explorations. I will then try to recommend a path of exploration for future that would aim to explore all the solvable issues, while trying to objectify the issues that can not or should not be solved at present with reasoning to support my decisions.

1. Keywords

Feature, Feature Interaction, Software Product Line, Feature Aware Verification, Product Line Verification, Feature Algebra, Feature composition, Model Checking, Variability, Feature Structure Tree, Super Imposition and Software Composition.

2. Introduction

In the modern world modularity is key ingredient in good development practices. While modularity in both code and ***software features*** is desirable and produces manageable software products within given time frames it also produces additional security concerns, the primary of which is the unintended characteristics that may be born off the interactions of these modular features known as ***feature interaction***. Thus a ***feature interaction*** is a situation in which the composition of multiple features leads to emergent behavior that does not occur when one of them is absent. While this problem has been studied thoroughly before it still persists as a major challenge for researchers. Multiple works on feature interaction detection have been carried out ranging from the use of ***superimposition*** (the practice of imposing various features on each other to see all possible characteristics) to ***Feature-aware verification*** (The detection of feature interaction based on specifications that do not have global feature system knowledge). While all works address some part of the problem, no comprehensive solution to the above mentioned problem statement has been developed yet.

Feature Interaction detection and elimination has a couple of major challenges: A first challenge, which was formulated by Hall, is to detect feature interactions based on specifications that do not have global system knowledge. The background is that the specification of a feature should not need to be aware of all other features of the system. It is desirable to specify and implement features in separate and composable units, while still being able to detect feature interactions.

A second challenge, which applies to product-line analysis in general, is to detect feature interactions without the need of generating and checking all individual products. Typically, many different feature combinations are possible, so detecting feature interactions by generating all possible combinations may not be feasible.

Since feature interaction detection and eliminations remains a fairly new area of research, it remains a area with multiple issues yet to be addressed satisfyingly.

3. Motivation

Feature-OrientedSoftwareDevelopment (FOSD) is a paradigm that provides attributes(formalisma, methods, languages and tools) that allow for the building of complex modular software systems.The main abstraction mechanism in FOSD is a feature.

Features are used to represent the requirement of the end user and per say are used to denote an increment in functionality. They are also used to clearly define the various components of a program or a software system thus helping in modularize all programs.

Feature Composition is thus the practice of composing code consistent with feature definition.

Research along different lines has been undertaken to realize the vision of FOSD. While most research paradigms agree on the common notion of Feature and Feature Composition, no common ground had been established in the case of techniques, representations and formalisms.

All modern programing paradigms are FOSD based. FOSD also forms the basis of the all modern good programming practices theory of modularity and building software in composable blocks.

In this paper I will try to relate the previously explored works and see how they as a single unit work to solve issues they were aimed at solving while exploring the individual contributions of the papers to the bigger problem statement. We will walk through the papers in the same order that we generated the previous readings and then will try to summarize their contributions as a single entity at the end of the section.

4. Approaches and State of Research

Feature Interaction Detection and Removal is a major problem that has been approached in multiple ways by each approach trying to solve the problem in the most general and optimized way possible.While some approaches aim to make the test systems more feature aware, others have tried to solve this problem by extracting meaning from the features themselves while maintaining their isolation properties and then solving for their interactions in a single pass. As we can see both these approaches are highly contrasting in nature.

In this section I will discuss the various techinques that have been presented and discuss the state of research in these areas:

4.1. Feature Interaction

Feature Interaction is defined as the generation of unintended characteristics in a software system when two or more features interact with each other. This uncharacteristic behaviour is non existent if a single component (or interacting feature) is missing.

Feature Interaction is an unintended characteristic in a software system and thus generally not good as it produces undesired effects that may have dire consequences such as the introduction of security risks born out of feature interaction.

Thus feature interaction detection and elimination is an important research problem and one that must be solved efficiently and effectively.

4.2. Software Composition

The process of composing software systems from the basic building blocks known as software artifacts is known as Software Compositions. The process essentially entails the use of modular blocks known as software artifacts which maybe combined to give different software systems depending upon the combination of software artifacts used.

Thus the above mentioned features are implemented as software artifacts. These software artifacts can then be combined in different ways to generate multiple different software systems.

While this combination of artifacts is desirable as it provides flexibility and reuse for software components, it also produces extraneous risks such

that may be introduced by unintended behaviours caused by the interaction of above mentioned artifacts.

This produces ***Feature Interaction*** as defined above.

4.3. Feature Structure Tree

One of the earlier works in detection and elimination of Feature interaction was introduced by Apel, Sven, Christian Kastner, and Christian Lengauer in their paper[1].

The defined Feature Structure Tree as follows: "A feature structure tree represents the essential modular structure of a software artifact and abstracts from language-specific details. An FST acomplishes this by the use of nodes to represent Language-specific charecteristics."

For example, an artifact written in Java contains packages, classes, methods, and so forth, which are represented by nodes in its FST; a Haskell program contains equations, algebraic data types, type classes, etc., which contain further elements; a makefile or build script consists of definitions and rules that may be nested.

Each node of an FST has 1) a name that is the name of the corresponding structural element and 2) a type that represents the syntactic category of the corresponding structural element.

Figure 1. Java code and FST of the artifact BaseDB, taken from the Berkeley DB case study.

For example in the above image, a Java class Foo is represented by a node Foo of type Java class. Essentially, an FST is a stripped-down abstract syntax tree: It contains only information that is necessary for the specification of the modular structure of an artifact and for its composition with other artifacts. The inner nodes of an FST denote modules (e.g., classes and packages) and the leaves carry the modules content (e.g., method bodies and field initializers).

4.4. Superimposition

The process of composing software systems by merging the substructure of two or more software artifacts is known as Superimposition. In this process a software system is essentially designed using merging of substructures of software artifacts as compared to simply combining the software artifacts to prepare software systems. Eg: If two JAVA files named Foo with same named classes, were to be combined, the result would still would be called Foo and all the internal functionalities of both the classes would be merged to have functionalities for both.

Superimposition has been applied successfully to the composition of class hierarchies in multi-team software development, the extension of distributed programs, the implementation of collaboration-based designs, feature-oriented programming, multidimensional separation of concerns, aspect-oriented programming,and software component adaptation.

Thus from above it stands to reason that ***Superimposition*** can be used to generate combinations of software artifacts to test for Feature Interaction.

Superimposition is only one of several composition approaches. It is especially useful in scenarios in which the code of components is available and their structures are compatible. Other scenarios such as black-box composition or the integration of structurally incompatible components are less suited for superimposition and should be handled by alternative composition approaches such as on-demand re-modularization and component aggregation.

The use of superimposition for software feature verification was proposed by Apel, Sven, Christian Kastner, and Christian Lengauer in their paper "Language-Independent and Automated Software Composition: The FeatureHouse Experience"[1] which introduced their tool "FEATUREHOUSE" based on Superimposition and Feature Structure Tree.

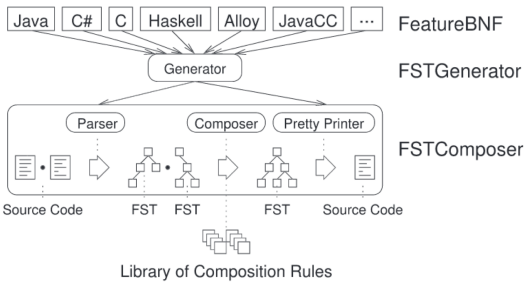

Figure 2. Architecture for FeatureHouse[1]

4.5. Model Checking in Product Line Engineering

"Model checking of domain artifacts means to verify that every possible product that can be derived from a domain artifact fulfills the specified properties." This essentially means that model checking is the method by which we check all the possible behaviours (possible reuses) of a domain artifact so that no fault is encountered when a domain artifact is reused. This concept has been proposed by Lauenroth, Kim, Klaus Pohl, and Simon Toehning in their paper "Model checking of domain artifacts in product line engineering."[2].

This thus is a counter paradigm to the superimposition paradigm proposed above[1]. This is also in contrast to model checking in single system development where a single product is verified if it fulfills the defined properties, model checking in product line engineering has to verify that a whole set of products fulfills the properties specified for each product. However, domain artifacts can not be verified by directly applying model checking approaches from single system engineering, since they do not take into consideration variability that is introduced due to the product line. The proposed solution by the authors is the introduction of an additional parameter known as ***Variability*** to be used alongside Model checking to take into account product line variability[2]. The next subsection will investigate this in more detail.

4.6. Variability and Its Use in Model Checking in Product Line Engineering

Product Line Variability has been defined by the authors by using the Orthogonal variability modeling language[3].

The orthogonal variability model provides variation points, variants, variability dependencies, and constraint dependencies to define the variability of a product line. Variation points can be segregated into mandatory variation points which ***have to be considered*** and optional variation points which ***might be considered if required.*** Variability dependencies define the allowed selection of a variant at a variation point. Differentiation is based upon mandatory (must be selected), optional (can be selected), and alternative (selection out of a defined set of variants). Constraint dependencies are used to further define constraints for the selection of variants and variation points. Differences are further fine grained into requires dependencies and exclude dependencies. A simplified example of a variability model is depicted in the upper part of Figure below.

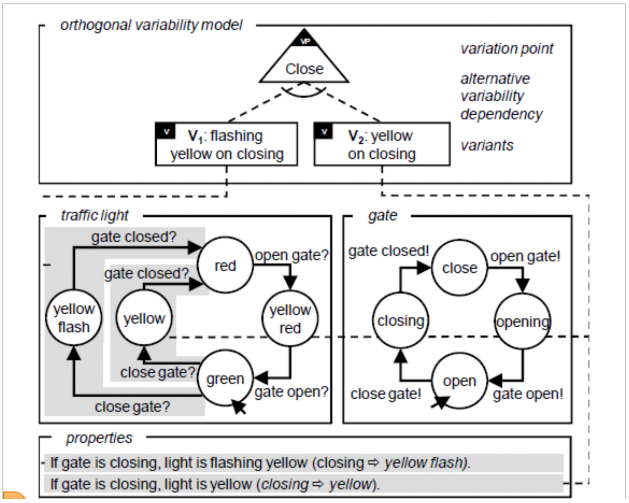

Figure 3. Simplified orthogonal variability model[2], two I/O-automata and two properties

If the variability model is ignored and only and only model checking approach from single system engineering is applied to the above demonstrated example presented in, model checking approach would state that both defined properties are not fulfilled by the specified system, since it is possi-

ble to reach the states (yellow flash, closing) and (yellow, closing) which are counterexamples for the validity of each properties.

However, this verification results is incorrect. The variability model does not allow to derive a product from the domain artifacts for which the property Formula is specified and which is able to reach the state (yellow flash, closing), or vice versa for which the property Formula is specified and which is able to reach the state (yellow, closing).

4.7. Feature composition

This can be defined as the process of composing the code to essentially implement the features as defined by the stakeholders. This step thus entails the actualy working implementation of the above mentioned feature. Code must be composed in a consistent way.

Different Researchers have proposed myriad ways of defining what a feature is [1][2][4]. One common definition accepted universally is: "A feature is a structure that extends and modifies a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer configuration options.[4]" This informal definition for feature composition has been used by Apel, Sven, Christian Lengauer, Bernhard Mller, and Christian Kstner[4] to define a formal framework for FOSD. Mathematically, the definitions proposed by the authors above may be represented as follows:

Assumption is an abstract set of features F and describe feature composition by the operator:

$*$:F X F $->$ F

Thus using this operator we can now define complex features as a group of more simplified features and a whole program maybe defined as the group of mathematically complex features.

Thus, mathematically using above definition, we may define a program as follows:

$$p = fn*(fn\text{-}1*(\ldots\ f2*(F1)))$$

The order of features in a composition may matter since feature composition is not generally commutative, and parenthesizing may matter since feature composition is not in every case associative, as we will show. For simplicity, we define feature composition such that each feature can appear only once in a composition. Allowing multiple instances of one and the same feature in a composition would be possible,but this would only complicate the algebraic framework and does not provide any new insights.

4.8. Feature Algebra

Feature algebra[4] serves a four fold purpose, namely:

- Abstraction: Used for abstraction from details related to Programming languages and environments in FOSD.
- Alternative design decision: Reflects variants and alternatives in concrete programming language mechanisms.
- Can be used for type checking and interaction analysis.
- To provide and architectural view of the software system.

This term has been introduced in An algebraic foundation for automatic feature-based program synthesis by Apel, Sven, Christian Lengauer, Bernhard Mller, and Christian Kstner[4] as original work and can be thought of as an extension of their original work in FOSD with use of Feature Structure Tree[1]. Feature Algebra has been defined by keeping the FST[1] at the base of it all.

Feature Algebra is essentially used to abstract programming concepts and for use as a concrete representation of features in programming language specific terms. Consequently, in this paper, the authors put forth the order for building their model based on the proposed Feature Algebra as follows:

- Defining the feature model;
- Defining the expression system for them (Feature Algebra);

- Defining some basic operations: composition, introductions, modifications, etc.;
- Finally all the above steps are combined to form the quark model[4].

This quark model is finally used for Feature Verification.

This paper thus essentially builds further on the earlier work presented by the same authors[1] where in the concept of the Feature Structure Tree was first introduced.

Feature algebra is thus a combination of FSTs[1] and a new concept called ***introductions*** that may be defined as: "Introductions are the abstract counterparts of FSF's. Therefore,every feature algebra has to comprise a set I of introductions. Among them one usually distinguishes a subset of atomic introductions.In the concrete algebra of FSFs these correspond to leaf nodes,characterized by the unique maximal paths from the respective roots to them.A basic feature can also be represented as the superimposition of all paths resp.atomic introductions in its FSFs. Hence,an abstract superimposition operator is the second main ingredient of a feature algebra;it is called introduction sum"

Thus they define Feature Algebra formally as:

"The feature algebra provides a formal foundation for FOSD. It abstracts from the concrete case of FSFs by listing the essential operators together with the algebraic laws, formulated as axioms, that we deem reasonable in some concrete setting of FOSD.All axioms hold in the concrete algebra of ordered FSFs as well as in the mentioned algebra of unordered FSFs, but also in many others, since they are not very restrictive. A manipulation of an algebraic expression induces a corresponding manipulation of an FSF"

4.9. Feature-Aware verification

This is a concept presented in "Detection of feature interactions using feature-aware verification"[6], paper by Sven, Apel; Speidel, Hendrik; Wendler, Philipp; von Rhein, Alexander; Dirk Beyer. This paper builds upon their older works which started with the generation of Feature Structure Trees and compositions, followed by the feature algebra which build on the concept of FST's and used them with Superimposition to make a mathematical definition for feature abstraction known as Feature Algebra[4]. They also add the concept of variability encoding for taking into consideration product line engineering introduced variability into their verification model known as variability encoding[3].

Feature Aware Verification is thus the technique of where a feature's properties are examined without the need of having domain specific knowledge of the rest of the software system.

This thus is the novel idea presented by Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, Dirk Beyer in their paper[6]. This is essentially a major step forward from the other older works based on superimposition and FST's as all those works need to generate all possible combinations of features interacting with one another to be effective, which for a large feature set was not a viable solution. Now in this case with the help of Feature Aware Verification Language specification introduced in this paper all features are examined individually in details and then *Product Line Verification*(defined below) is used to detect all feature interactions in a single pass.

4.10. Product Line Verification

Sven Et. Al.[6] have suggested the use of product line simulators that contains all features of a product line for product line verification. In these simulators all features can simply be toggled as per requirements. The state space of the product simulator subsumes the state spaces of all valid products of the product line such that the model-checking procedure can benefit from it during the checking process. It allows the model checker to detect feature interactions more efficiently, because not all individual feature combinations have to be unfolded in the model checker's state space. Thus all model checking can be carried out in a single pass instead of the more complex brute force method.

The product simulator is obtained by variability encoding. The procedure of variability encoding is a modification of the regular composition process in order to create the product simulator incorporating all features of the product line. First, variability encoding defines for each feature a global boolean feature variable that models the presence or absence of the feature. Second, variability encoding introduces for each function refinement a dispatcher function that dispatches between the refined and the refining function depending on whether the feature that contains the refinement is selected. Third, variability encoding stores the dependency constraints between features (i.e., the feature model) using a boolean predicate over the boolean feature variables. Finally, the program execution is encapsulated in a conditional block that is executed only if the constraints imposed by the feature model are satisfied; this way, execution paths that are associated with invalid feature combinations are not considered by the model checker.

After variability encoding, we check the resulting product simulator against the specification of all features of the product line. We initialize the boolean variables of the features using a nondeterministic choice so that the model checker must assume that all feature combinations defined by the feature model may occur. This way, the model checker checks all possible feature combinations, that is, all combinations of feature code, without generating all individual products.

4.11. Software Product Lines and Feature Sensitive Dataflow analysis

Software Product Lines: Software product lines (SPLs) are commonly developed using annotative approaches such as conditional compilation that come with an inherent risk of constructing erroneous products. For this reason, it is essential to be able to analyze SPLs. A software product line (SPL) is a family of related software products sharing a common set of assets. Differences and commonalities between products are typically described in terms of features. Users customize a product by means of a selection of features that satisfies their functional requirements

Any dataflow analysis solution consists of three main components:

- ***Control Flow Graph***: Data flow analysis is carried out on a CFG. CFGs are used as abstract representations for programs on which ***Dataflow analysis*** needs to be carried out. CFG's are essentially directed graphs where nodes represents statements in the program and edges represent the flow of control.

- ***Lattice***: Latice in its essence is used to represent values of interest from a dataflow analysis of the program. Thus information calculated in a data flow analysis is stored in a Lattice $L = (D, \subseteq), where D is a set of elements and \subseteq is a$ ***partial-order*** $on the elements.$

- ***Transfer Functions:*** Transfer Functions are functions that simulate run time conditions (execution) at compile time. The transfer functions are associated to each individual statements and are used to simulate execution condition for that individual statement at compiler time.

Below is a diagram representing all stages of Dataflow analysis:

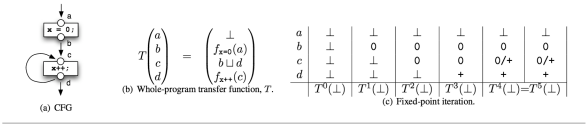

Figure 4. Stages of DataFlow Analysis

Claus Brabrand et. al. propose a new method for using control flow analysis, essentially a Compiler Related method for Software Product Line verification by the use of Feature-Sensitive Dataflow Analysis. They propose to make dataflow analysis Feature Sensitive by the following:

- Control-Flow Graph: In addition to the basic CFG's the authors also associate a set of configurations, for which the nodes corresponding statement is executed. They reffer to the process as CFG instrumentation.

- Latice: No changes are needed in the lattice for feature sensitive data flow analysis as configurations does not change the lattice.

- Transfer Function: The only major change needed here is to associate the configuration set to the original transfer functions to make this process Feature-Sensitive.

Further the authors propose Consecutive and Simultaneous Feature Sensitive Dataflow Analysis using the above methods.

Thus using the above techniques software product line verifications may be conducted using Feature Sensitive Dataflow Analysis.

4.12. SPL Conqueror

Norbert Siegmund et. al. propose a new tool called SPL conqueror for Software Product Line Verification. In this they propose the analysis of functional paradigms of a feature while treating the feature itself as a black box.

5. Conclusion

In conclusion I can say that the both major directions of research, namely, Dataflow analysis based and FST based, seem promising. While a large amount of work has been done in Dataflow analysis for program analysis, if the work is directly or partially translatable to Software Product Line Analysis it promises large benefits and quickly. This make this direction of research highly lucrative.

On the other hand the FST based approach is relatively new and has been developed with keeping this eact problem in mind and thus may be better suited to the requirements of our problem statement.

Thus in conclusion I would like to say that I find both lines of research highly lucrative, though from a personal stand point I would prefer to explore the area of Dataflow analysis because if translatable to our problem statement it would further research in the area at a more rapid pace.

Lastly I would like to point out, since quantative results are not relavant and research methodoligies are evaluated based on experimental results it is difficult to measure the various methodoligies head to head.

6. References

1. Apel, Sven, Christian Kastner, and Christian Lengauer. "FEATUREHOUSE: Language-independent, automated software composition." Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009.

2. Lauenroth, Kim, Klaus Pohl, and Simon Toehning. "Model checking of domain artifacts in product line engineering." Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on. IEEE, 2009

3. Apel, Sven, Christian Lengauer, Bernhard Mller, and Christian Kstner. "An algebraic foundation for automatic feature-based program synthesis." Science of Computer Programming 75, no. 11 (2010): 1022-1047

4. Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, Dirk Beyer. 2011. Detection of Feature Interactions using Feature-Aware Verification. Proceeding ASE '11 Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering

5. R. Hall. Fundamental Nonmodularity in Electronic Mail. Automated Software Engineering, 12(1):4179, 2005

6. H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In Proc. ASE, pages 347350. IEEE, 2008

7. S. Apel, C. Kstner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In Proc. ICSE, pages 221231. IEEE, 2009

8. Analysis Strategies for Software Product Lines by Thomas Thm, Sven Apel, Chris-

tian Kstner, Martin Kuhlemann, Ina Schaefer, Gunter Saake.

9. S. Apel, C. Kstner, and C. Lengauer. Strategies for product-line verification: case studies and experiments. Proceeding ICSE '13 Proceedings of the 2013 International Conference on Software Engineering Pages 482-491 IEEE Press Piscataway, NJ, USA 2013

10. A Classification and Survey of Analysis Strategies for Software Product Lines Thomas Thm, Sven Apel, Christian Kstner, Ina Schaefer, Gunter Saake.

11. H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. In Proc. FSE, pages 89-98. ACM, 2002

12. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Emerson and Aravinda Sistla, editors, Computer Aided Verification, volume 1855 of Lecture Notes in Computer Science, pages 154-169. Springer Berlin / Heidelberg, 2000. 10.1007/10722167-15.

13. Peter Puschner. Guest Editorial A Review of Worst-Case Execution-Time Analysis. Real-Time Systems, 18(2):115-128, 2000.

14. T. Thum, I. Schaefer, M. Kuhlemann, and S. Apel. 2011b. Proof composition for deductive verification of software product lines. In Proc. Intl Workshop Variability-intensive Systems Testing, Validation and Verification (VAST11). IEEE, Washington, DC, 270277

15. S. Apel,W. Scholz, C. Lengauer, and C. Kstner. Detecting Dependences and Interactions in Feature-Oriented Design. In Proc. ISSRE, pages 161-170. IEEE, 2010.

16. Delaware B.; Cook, W.R.; Batory D.: Fitting the pieces together: A Machine-Checked Model of Safe Composition. In Proc. of ESEC-FSE 09, 2009

17. Grumberg, O.; Veith, H.: 25 Years of Model Checking. LNCS Vol. 5000, Springer, 2008

18. Larsen, K.; Nyman, U.; Wsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In Proc. of ESOP, 2007, pp. 64-79.

19. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption Norbert Siegmunda, Marko Rosenmllera, Christian Kstnerb, Paolo G. Giarrussob, Sven Apelc, Sergiy S. Kolesnikovc