Do #ifdef-based Variation Points Realize Feature Model Constraints?

Alcemir Rodrigues Santos
Reuse in Software Engineering Laboratory
Federal University of Bahia
Salvador, Brazil
alcemirsantos@dcc.ufba.br

Eduardo Santana de Almeida
Reuse in Software Engineering Laboratory
Federal University of Bahia
Salvador, Brazil
esa@dcc.ufba.br

DOI: 10.1145/2830719.2830728

ABSTRACT http://doi.acm.org/10.1145/2830719.2830728
Two mechanisms widely used in the Software Product Lines (SPL) Engineering are the feature model and the conditional compilation. The former models the variability in the problem space and the latter realizes it in the solution space. Even though the research community know that the feature model imposes a number of constraints to the product line implementation, there is a lack of support to co-evolve problem space and solution space. In this paper, we present an exploratory study whether problem space constraints are considered at source code level of `#ifdef`-based SPL implementations. In order to accomplish our goal, we developed a preliminary approach to check problem and solution spaces in a prototype tool (FCLCHECK). The results show a lack of realization of feature model constraints while implementing variation points with that mechanism. We also evaluated the *scalability* of the approach and the *recoverability* of the tool.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms
Sofware Product Lines Engineering

Keywords
Software Product Lines Engineering, Conditional Compilation, Consistency Checking.

1. INTRODUCTION
Feature models are tree-like menus of configuration options, which may have cross-tree constraints among the features. While they model the variability at high level of abstraction, `#ifdef`'s directives commonly achieve it at source code level. In fact, among the variability realization techniques, conditional compilation still plays a major role in SPL engineering. For instance, `Linux` and `eCos` use conditional compilation as a mechanism to realize variability.

To maintain the consistency between the feature model and the source code on such large systems using `#ifdef`'s directives might become a problem. In fact, supporting the consistency among the software artifacts is a known issue in the context of software evolution [4] and it is still a major challenge [7]. In the SPL context, different work [3][5] addressed the evolution `#ifdef`-based systems. Feature models are not immune to inconsistencies [7], which demands for co-evolution of feature models and source code support. To the best of our knowledge, only Le *et al.* [2] investigated the consistency between feature models and its variation points' (VP) implementation. However, their approach is characterized by extracting the feature model from source code disregarding possible inconsistencies that may exist throughout the codebase. Therefore, there is a lack of research addressing consistency directly in `#ifdef` VPs.

This paper presents an exploratory study on the consistency between feature models and its `#ifdef`-based implementation. We analyzed four open source large systems, namely, `busybox`, `uClibc`, `eCos`, and `Linux`. Both, feature models and their constraints of all these systems are available [5]. Our results show that developers do not take feature models constraints into consideration while programming. Besides, we show our approach to identify such inconsistencies can scale for large systems within reasonable time.

We organized the remaining of this paper as follows. Section 2 goes deeper in the discussion about consistency between problem and solution spaces. Afterwards, Section 3 describes our study settings, the followed methodology, the tool support, and the target systems. In Section 4, we present and discuss the results of the exploratory study. Section 5 describes the study limitations, as well as, the lessons learned. Section 6 presents related work. Finally, Section 7 concludes the paper and presents future work.

2. PROBLEM STATEMENT
In this section, our aim is *(i)* to describe some inconsistencies that may arises in the development phase of the source code assets and *(ii)* to expose neglected variability implementation through examples.

2.1 Cross-tree constraints Implementations
Cross-tree constraints relate a feature `F` to a feature group `G` (composed of one or more features). In the sense that, if the application engineer selects `F`, the constraint may indicate the need of inclusion or exclusion of `G`. There are three well-known constraints rules: *(i)* `includes/requires` (when selecting `F` means the engineer must also selects `G`); *(ii)* `excludes` (when selecting `F` means the engineer must excludes `G`); and *(iii)* `mutual exclusive` (when selecting `F` means the engineer must excludes `G` and vice-versa) [1]. Some observations arise when taking such cross-tree constraints into consideration while developers are implementing variability with conditional compilation. Table 1 summarizes them.

Without considering such observations, developers are introducing variability implementation inconsistencies and product variants are being exposed to faults, which may be unacceptable. Next, we discuss the possible inconsistencies that may result from the neglecting feature constraints in the `#ifdef`-based implementation.

2.2 Implementation Inconsistencies

Table 1: Logical rules observations for implementing cross-tree constraints.

F **includes** *G*: by the time of the implementation of the feature `F`, developers may use "`#ifdef F`" only and disregard the cross-tree constraint existent. However, it is reasonable to think about taking the constraint into consideration and declare "`#ifdef (F && G)`" instead, to avoid faults in future product configurations.
F **excludes** *G*: similarly, to implement the variability imposed by `G`, developers should consider the `F` constraint and use `#ifdef (!F && G)` instead of `#ifdef G`, only.
F **mutual-exclusion** *G*: this constraint produces the same observation for the "excludes" cross-tree constraint applied for both, `F` and `G`. Developers should observe the constraint in both ways depending on which variability is being implemented.

Inconsistencies may be introduced in the code as highlighted before. Our goal is to show the importance of taking constraints into consideration rather than build an exhaustive catalog of inconsistencies. Table 2 enumerates some of these inconsistencies.

3. STUDY SETTINGS

The main goal of this study is to identify whether the feature model constraints are realized in the source code or not. Thus, *inconsistency* throughout this paper means the absence of features with constraints related in the variation points implementation. Therefore, three objectives guided the investigation: (**O1**) to *characterize* such inconsistencies; (**O2**) to evaluate the *scalability*; and the *recoverability* of our consistency checking approach (**O3**). Both, **O2** and **O3** are auxiliary objectives and we expect they indicate that the consistency checking approach can be extended to support co-evolution of large SPLs. We call recoverability because we do not have a ground truths in terms of which are the inconsistencies existent in the code. Next, we describe *(i)* the followed *methodology*, *(ii)* the proposed *tool support*, and *(iii)* the chosen *target systems* used in this exploratory study.

3.1 Methodology

Figure 1 summarizes the defined methodology. First, we collected both problem and solution space assets: cross-tree constraints available from a previous study [5]. Afterwards, we used our consistency checking prototype tool (named FCLCHECK) *(i)* to collect some metrics from the inconsistencies identified in the target systems to perform their characterization and *(ii)* to measure the time consumed in the checking tasks.

O1 was addressed in a twofold way: *(i)* by calculating the occurrences of each constraint dependency type among the inconsistencies to discover which is more likely to happen and *(ii)* by calculating the average of inconsistencies per variation points and per constraints. To address **O2**, FCLCHECK calculated the time consumed to execute its tasks. Inconsistencies were introduced purposely to address **O3**. A Ph.D. student played the system developer's role and introduced inconsistencies by pasting `#ifdef <feature-tag> #endif` into the original code for each target system. The `<feature-tag>` had to have an associated constraint in order to characterize the inconsistency.

3.2 Tool Support

The tool supporting this experimental study is named FCLCHECK[1]. It relies on static analysis of the source to detect inconsistencies on the variability point's implementation. Figure 2(a) shows an overview of the approach for consistency checking. A variability analysis actor (*e.g.*, an application engineer or a variability analysis tool[2]) takes the "Feature Specifications" and the "Source Code" to specify the "Feature Constraints". Alternatively the constraints may be extracted from feature models. Afterwards, FCLCHECK identifies the "Feature Implementation Inconsistencies".

3.2.1 Implementation

FCLCHECK is an Eclipse plug-in that automates the consistency checking between feature constraints extracted from the feature model (problem space) and `#ifdef`-based implemented variation points (solution space). Specifically, FCLCHECK checks whether the implemented variation points using conditional compilation (*e.g.*, `C` preprocessor) realize the constraints defined in the feature model (*e.g.* LVAT constraints files).

FCLCHECK performs the consistency checking in two different ways: by 'full building' (when all the source files are checked) and by 'incremental building' (which checks only the changed artifacts). The inconsistencies found by the tool are shown in the source code editor (Figure 3(a)) – as a line mark where FCLCHECK found a possible inconsistency – and in the 'Inconsistencies View' – which allows the user to open a given file to review the code where the inconsistencies were found (Figure 3(b)). Basically, FCLCHECK's implementation follows an architecture with four modules: Builder; Parser; Validator; and Output. Table 3 describes each one of them.

3.2.2 Finding Inconsistencies

In this study, we are interested on the identification of consistency violation by absence, *i.e.*, to find which variation points are neglecting the constraints imposed by the feature model. In this sense, Figure 2(b) describes the FCLCHECK 3-step algorithm. In *Step 1*, it parses the source code mapping the `#ifdef` blocks. In *Step 2*, it parses the dependency constraints input. Finally, in *Step 3*, it checks features used in the VPs against the dependencies mapped.

Currently, FCLCHECK performs the consistency checking as follows: *(i)* it checks whether a constraint concerning `FEATURE` exists; if so, *(ii)* it produces an inconsistency mark in the `#ifdef` declaration to each associated constraint. The constraints dependency types addressed in this study are: `includes/requires`, `excludes`, and `conditional inclusion` (if, and only if).

3.3 Target Systems

We choose four highly configurable open-source projects using the `C` preprocessor mechanism to implement variability to perform our study. Namely: *(i)* **Linux x86** 2.6.33.3 – which has 7,691 source files and 6,559 features; *(ii)* **uClibc x86_64** 0.9.33.2 – which has 1,628 source files and 367 features; *(iii)* **BusyBox**

[1]Available at `http://github.com/alcemirsantos/fclsuite`

[2]For instance, LVAT, available at `http://bit.ly/1R9ZJUJ`

Table 2: Examples of inconsistencies that may be introduced.

Implementing an optional instead of a XOR feature. The developer may neglects the `excludes` constraints existent in the XOR-group and implements an alternative feature as an optional one. For instance, declaring `#ifdef A`, only.
Adding cross-tree constraints instead of implement OR feature. Developers might implement such feature group by using annotations like "`#ifdef ((A&&B) || (B&&C) || (C&&A))`". Although this implementation can be correct, there is a chance the developer has introduced 'cross-tree' constraint among the OR group features.
Relaxing cross-tree constraints. Considering that feature `A` *includes* a feature `B` or is *excluded* by `C`. Such constraint may be relaxed if the developers use the `#ifdef A` strategy to implement `A`.

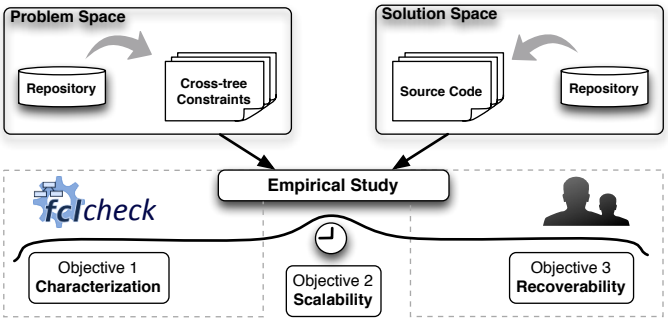


Figure 1: Empirical study methodology.

Table 3: Logical rules observations for implementing cross-tree constraints.

Builder: it *(i)* launches the `Parser` module and *(ii)* decides whether to use *full* or *incremental* build mode.
Parser: It parses the constraints input and the source code to allows the *Validator* module to compare variation points (VP) against constraints. Currently, it is able to extract constraints from LVAT constraints file, only.
Validator: It compares VP against dependency constraints. Currently, it checks VPs with one feature only (*e.g.*, `#ifdef ANY_FEATURE`). In addition, it only checks 2-feature constraints, rather than constraints with logical expressions.
Output: This module presents the violations detected by the validator module both in the source code editor as a mark in the variation point (Figure 3(a)) and in the 'Inconsistencies View'. (Figure 3(b)).

1.21.0 – which has 535 source files and 921 features; and *(iv)* **eCos i386** 3.0 – which has 579 source files and 1,254 features. All of them are large systems and their variability models have been created, maintained, and evolved by the original developers of the systems over periods of up to 13 years. Therefore, we believe these systems represent a large number of variability models and code-base sizes. Moreover, these systems have variability models available and they were already used in a previous study within this context [5]. Such choice can guarantee a reliable source of information to our study and strengthens the observations.

4. RESULTS AND DISCUSSION

In this Section, we discuss the results achieved in our empirical study regarding each objective (Figure 1): (O1) inconsistencies characterization; (O2) approach scalability; and (O3) tool recoverability.

4.1 O1: Inconsistencies Characterization

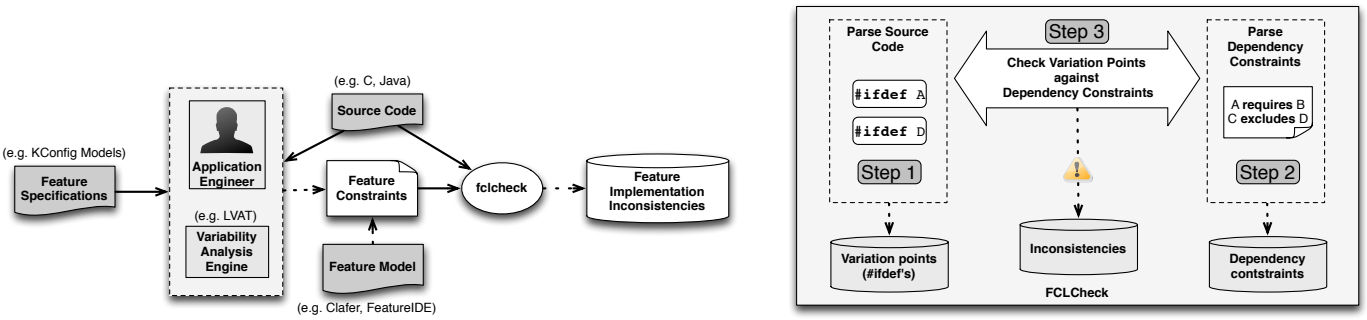
We collected four different metrics to perform the characterization: *(i)* total number of inconsistencies found, *(ii)* the number of inconsistencies by each variation point, *(iii)* the number of inconsistencies by each constraint, and *(iv)* the number of inconsistencies by each constraint dependency type. Table 4 shows these values for each target system.

Table 4: Metrics collected.

System	Inconsistencies					
	# VP	#C	# Dependency type			#Total
			i	*e*	*ci*	
uClibc	.85	14.87	262	292	338	892
BusyBox	1	.03	3	5	0	8
eCos	1.3	3.4	645	140	197	982
Linux	.35	5.63	15752	10756	10200	36708

VP: Variation Point; **C:** Constraint; **i:** includes type; **e:** excludes type; **ci:** conditional inclusion.

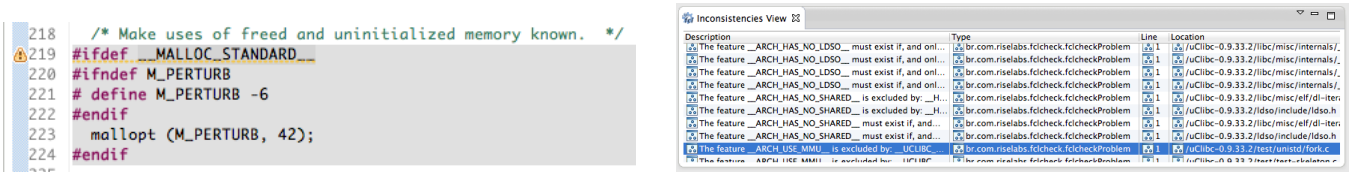
No pattern emerged from the metrics calculated, perhaps the different nature of each system, as well as, their sizes and characteristics contributed for this fact. However, such a huge number of



(a) A FCLCHECK approach overview.

(b) 3-steps consistency checking algorithm.

Figure 2: Average time spent in each task.



(a) Inconsistency Mark

(b) Inconsistencies View

Figure 3: FCLCHECK reporting of inconsistencies.

inconsistencies identified (*e.g.*, `Linux`) suggest that feature model constraints are not taken into consideration while implementing the variation points. We believe this happens because developers rely on external tools to implement the existing constraints. For instance, in case of developers use an external variability management approach, such constraints may be captured somehow. Other explanation is that developers can avoid expressing such constraints at code level by a project decision, since large SPL with complex constraints would become harder to maintain.

4.2 O2: Scalability

With regarding to the scalability of the FCLCHECK algorithm, we took all FCLCHECK sessions in the same computer, Intel Core i5 2.3GHz processor and 16GB of RAM memory (1333 MHz DDR3). Table 5 shows the time elapsed while consistency checking the target systems in *full build* mode. All time values were measured in seconds. To fully check the `Linux`, which is by far the biggest target system, FCLCHECK took about a half past four hours (with time to do statistics included – which takes very long because was necessary several iterations in large arrays). In fact, for *incremental build* where only a delta of the changed files is checked, it takes much less time; in fact, it is almost instantly.

Table 5: Time elapsed while consistency checking the target systems in *full build* mode.

System	Average Time (s)		Total Time (s)
	# per VP	# per Resource	
uClibc	$\approx .9 \times 10^{-2}$	$\approx 1.3 \times 10^{-2}$	≈ 51.22
BusyBox	$\approx 2.7 \times 10^{-2}$	$\approx 3.3 \times 10^{-2}$	≈ 26.42
eCos	$\approx 2.8 \times 10^{-2}$	$\approx 6.6 \times 10^{-2}$	≈ 137.72
Linux	$\approx 4.7 \times 10^{-1}$	$\approx 7.3 \times 10^{-1}$	$\approx 19.1 \times 10^{3}$

VP: Variation Point; **s:** seconds

The fact that the `Linux kernel` has a lot of larger files than the other systems contributes to the huge difference on the building time. However, even `eCos`, which is another operating system, took around two minutes to build and approximately 138 times less time than resources from `Linux` to be checked. The other two systems `BusyBox` and `uClibc` are smaller and they took less than a minute to build the entire project and milliseconds to build each resource. These values show that our approach scales to real word sized product lines.

4.3 O3: Recoverability

The results of the identification of inconsistencies in the variation points introduced by the fake developer were interesting. The tool added different inconsistency marks to the variation points introduced by the fake developer. These marks emerged either from the constraint the fake developer choose or from other constraints existent in the model. The inconsistencies introduced had the correctness of the FCLCHECK output manually checked. Our tool was able to correctly assign the inconsistency mark to all the inconsistencies introduced by the fake developer.

5. LIMITATIONS AND LESSONS LEARNED

Someone can consider that FCLCHECK does not cover most SPL constraints since they can be much more complex than a two feature relationship. However, She *et al.* [8] found that 89% of the constraints existent in Linux are positive implications (also known as `include/requires` constraints). Therefore, FCLCHECK can identify a large number of the inconsistencies of a wide range of systems.

FCLCHECK only detects "absence constraint violations" — when some feature status check `isEnabled(FEATURE)` is missing on the variation points implementations — and the checking of "divergence constraint violations" is uncovered yet. To check such violations requires additional semantic analysis of the logical expressions defined in the variation point's implementation, which is not a trivial task.

Regarding the empirical study, there is some threats to validity. Despite the attempt of execute only the consistency checking related processes, the time measured might be inaccurate since the computer was not processing only FCLCHECK's tasks. Furthermore, only one fake developer seeding the inconsistencies can also be seen as a threat.

While conducting this study, we discussed on the importance of supporting co-evolution of both assets: feature model and its implementation. To realize feature model constraints at source code level seems to be undesirable either for maintainability reasons (*e.g.*, it increases the complexity and code-obfuscation) or to avoid the cost of maintain redundancy on the consistency assurance (*e.g.*, variability management approach and code level consistency).

6. RELATED WORK

In a previous work [7], we identified the literature addressing the consistency between source code and models. Next, we highlight some of them.

Tartler *et al.* [9] addressed variability implementation inconsistencies between the *KConfig* model and `C` preprocessor directives blocks by identifying dead blocks in the Linux build system. Le *et al.* [2] extracted feature models from code and check against an existing one to validate the consistency of the implementation. Passos *et al.* [6] identified evolution patterns among co-related artifacts of the `Linux`.

None of this work directly addresses the co-evolution of the problem and solution spaces together. Terra *et al.* [10] present a recommendation system to guide developers on the system evolution without violate any architecture rule defined, *i.e.* they promote software evolution by maintaining the architectural conformance. Their approach relies on a domain specific language that helps the software architect to define the possible behavior and relationship among the different modules in the system. In addition, their approach also uses the Eclipse infrastructure to implement such support on Java-based software systems. This work highly inspired us on the inception of the defined approach.

7. CONCLUDING REMARKS

In this study, we conducted an exploratory study to identify whether developers of open-source `#ifdef`-based systems take feature models constraints into consideration in the development phase. Indeed, our prototype tool (FCLCHECK) identified a number of inconsistencies in the variation points regarding the feature model constraints of each addressed target system. This may indicate that developers rely on external configuration tools to satisfy constraints imposed by the feature models rather than expose those constraints in the variation points. In addition, this study indicates that `include/requires` dependency constraints are more likely to produce inconsistencies.

Additionally, another contribution of this work is the preliminary support for co-evolution of feature models (problem space) and its implementation (solution space) during the development phase, which was implemented in the prototype tool FCLCHECK. In this study, we also showed the tool can scale to large systems with reliable results regarding the proposed consistency checking approach.

As future work, we intend to address the limitations of the tool by improving its functionality and by covering additional consistency checking activities, such as, by handling complex variation points declared with logical expressions. Additionally, further investigation is needed to understand whether there is a correlation between the consistencies and eventual bugs reported in the issue trackers of the systems.

Acknowledgment

The authors would like to thank to *Raphael Pereira de Oliveira* and *Ivan do Carmo Machado* for the valuable help in the former drafts of this paper. This work was parcially funded by FAPESB and INES[3].

References

[1] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[2] D. Le, H. Lee, K. Kang, and L. Keun. Validating consistency between a feature model and its implementation. In *Proc. of the Int'l. Conf. on Soft. Reuse*, volume 7925, pages 1–16. 2013.

[3] F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonseca. A catalogue of refactorings to remove incomplete annotations. *Universal Computer Science*, 20(5):746–771, 2014.

[4] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Proc. of the 8th Int'l. Work. on Principles of Software Evolution*, pages 13–22, 2005.

[5] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proc. of the 36th Int'l Conf. on Software Engineering*, pages 140–151, 2014.

[6] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *Proc. of the 17th Soft. Product Line Conference*, pages 91–100, 2013.

[7] A. R. Santos, R. P. de Oliveira, and E. S. de Almeida. Strategies for consistency checking on software product lines: A mapping study. In *Proc. of the 19th Int'l. Conf. on Evaluation and Assessment in Soft. Eng.*, 2015.

[8] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Variability model of the linux kernel. In *Proc. of the 4th Int'l. Work. on Variability Modeling of Software-intensive Systems*, 2010.

[9] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *Software Tools for Technology Transfer*, 14(5):531–551, 2012.

[10] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, 39(12):1073–1094, 2009.

[3] More information available at `http://www.ines.org.br/`