



Hildesheimer Informatik-Berichte

Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid

# Mismatched Configuration Information of Linux

**October 19, 2016**

Report No. 1/2016, SSE 1/16/E

Software Systems Engineering • Institut für Informatik
Universität Hildesheim •Universitätsplatz 1 • D-31134 Hildesheim

**Abstract**

**Context.** Software product line engineering has been established to minimize costs and efforts, while maximizing the quality of products in a family of software products. Software product lines typically contain a variability model, which supports the derivation of permissible variants. These variability models may contain expert/domain knowledge in the form of constraints. These constraints are used during the configuration process to avoid the selection of unsupported product variants.

**Problem.** Developers must encode their knowledge about supported product variants and restrictions, otherwise the variability model becomes ineffective or even incorrect. The initial development of the variability model as well as the evolution of the product line implementation bear the risk that model and implementation drift apart. In this report, we introduce the notion of *mismatched configuration information* to describe the situation if the variability model does not reflect the dependencies of the implementation. This may indicate an incomplete variability model or undesired dependencies between code artifacts.

**Solution.** We discuss the impact of mismatched configuration information and show how to detect this conceptually. Subsequently, we focus on mismatched hierarchical configuration information and present an effective heuristic for their detection. These results serve as an input to complete variability models or a code review to remove undesired implementation dependencies. We discuss the application of our approach on a Linux case study. The analysis of the x86 architecture of the Linux kernel takes only around 30 minutes and revealed mismatched configuration information, which was not treated by prior work.

# Contents

# List of Figures



# List of Listings



# List of Tables

# 1 Introduction

Software Product Lines (SPLs) are inherently significantly more complex than individual systems. This increase in complexity is due to the need to cover all relevant variations. In addition, product lines typically contain a variability model, which supports the derivation of the permissible variants. In particular, the variability model is supposed to represent all necessary knowledge on which features can be combined. However, often the product line implementation itself also restricts the potential products in a more or less explicit way. This distribution of dependencies to (at least) two different artifact types results in a situation where the actual situation may become unclear or even inconsistent.

New concepts have been developed to address the challenges of quality assurance in the context of SPLs. For instance, testing strategies of variable code artifacts [vdLSR07, ABKS13] or the detection of configuration anomalies in feature models like *dead features* [MWC09, BSRC10], which are leading to misinterpretations of the variability model [vdML04]. Recent research addresses the combination of information of the variability model and the code artifacts to facilitate static analysis of product line assets [TLSSP11, LvRK+13]. However, this work targets the consistency of the different artifacts. In this report, we present an approach for checking the completeness of the variability model to support users during the configuration process, as demanded by Jean Delvare, a maintainer of the SUSE distribution [Del14].

First, we discuss the impact if the variability model misses information of implemented dependencies. Instead of the detection of all divergences of the variability model and the implemented artifacts, which may overwhelm the developer with too many results, we focus on the efficient detection of a specific kind of mismatched configuration information: *missing nesting dependencies*. We expect that these results will be more comprehensible to developers as they represent locally hierarchical structures of the implementation and do not cover widely scattered cross-tree dependencies.

This report is structured as follows: in the next section, we discuss related work on analysis methods for detecting variability-related issues in the context of software product lines. In Section 3, we discuss the concept of mismatched configuration information and their impact. In Section 4 we focus on the detection of a particular class of mismatched configuration information, which we call *missing nesting dependencies*. Section 5 describes the implementation of our approach. Section 6 evaluates our approach based on the Linux kernel. Finally, Section 7 provides conclusions and outlines some future work.

# 2 Related Work

In this report, we compare the variability information of code assets with the modeled variability information to detect potential problems in a software product line, which may be seen as an instance of variability smells [ABKS13]. We identified two categories of papers, which are relevant for this work: similar *approaches for detecting variability errors*, and existing *analysis tools*, which are appropriate for this kind of problem detection. The remaining section is organized according to these two categories.

**Approaches for detecting variability errors**. In recent years, much effort was spent to detect defects related to software product line development, like (un-)dead features [BSRC10], (un-)dead code [TLSSP11], or variability-aware type checking and liveness analysis [LvRK+13]. Also Thüm et al. [TAK+14] survey general analysis strategies for software product lines. Even if the main objective of these analysis techniques is the detection of logical defects, some techniques also include the detection of further issues, e.g., detection of superfluous code [TLSSP11] to resolve maintenance issues, a form of a technical debt [Sch13, KNO12].

Nadi et al. [NBKC14, NBKC15] present an approach for the automated extraction of variability model constraints based on a variability-aware code parser [KGR+11]. They aim at support for creating variability models instead of a consistency check or smell analysis. They evaluated their approach among others based on the Linux kernel and were able to extract approximately 33% of the hierarchical constraints and 5% of the cross-tree constraints. They argue, that the missing constraints address in large part expert knowledge or run-time behavior, which cannot be detected by a statical analysis.

**Analysis tools**. Most of these software product line analysis approaches have been evaluated based on the Linux kernel and similar open source projects. For this reason, various research prototypes were developed, which are able to handle the variability information of the Linux kernel. Most of these tools are able to translate the variability information into propositional logic, which may be processed by SAT-solvers. The authors of these research prototypes made their tools public available to facilitate the replication of their experiments as well as to support the reuse of their tools in different contexts. In [ESKS15, ESKAS15], we analyzed the quality of tools regarding the translation of Kconfig models, a textual language used to manage the variability of the Linux kernel. This translation is not a trivial task, since the semantics are rather unclear because of many corner cases. In the implementation (cf. Section 5) we reuse some of these tools based on our comparison [ESKS15].

# 3 Mismatched Configuration Information

In this section, we introduce the notion of *mismatched configuration information*, before we continue in the next section with a detailed analysis of a specific form: *missing nesting dependencies*.

## 3.1 Problem Description

Mismatched configuration information describes a divergence between configuration information in the variability model and implemented variability of (code) assets. This may hamper the configuration process or lead to errors during product derivation. For example, the following defects may be introduced through mismatched configuration information:

**Redundant Configurations**
: The derivation process of a correct product line should lead to different product instances for different configurations. As a result of mismatched configuration information, this may be violated. While this may be intentional, it may also be created as an accidental problem, especially if the variability information is distributed across a number of different artifacts (e.g., variability model, code, build model). This problem is also mentioned by Jean Delvare, a maintainer of the SUSE distribution [Del14].

  Also the authors of [NBKC15] argue that all valid configurations should lead to lexically different products. More precisely, the manual selection of a feature $f$ should affect the product derivation. If the feature $f$ has no effect unless other features are selected, it should automatically be disabled or hidden to simplify the configuration process. The authors call this *feature effect*. In a detailed analysis of the x86 architecture of the Linux kernel v2.6.33.3, the authors detected 219 dependencies[1], which are not covered by the variability model.

**Erroneous Code Derivation**
: Additional or incorrect dependencies may lead to the derivation of semantically incorrect products or at least products that differ from the semantics intended by the configuration. This is a rather critical defect as the configured product is compilable and hence the issues are not initially visible.

**Dead or Undead Features or Code**
: This has received significant attention in the scientific literature [TLSSP11, BSRC10]. It has been mostly treated as an error, but when analyzing where these errors stem from, this may be due to complexity of the variability as described by Tartler et al. in [TLSSP11] in their example for the concept of "logic integrity violation".

[1] Detailed measuring results are available at `https://bitbucket.org/snadi/farce-linuxanalysis/raw/365342ccb32bd51c05d41c0646841d8e89bf380c/output/Comparison/CodeVsFeatureModel/linux_accuracyStats.csv`

**Potential for Evolutionary Problems**

Missing configuration information may be compensated through additional code dependencies to avoid miss-configured products. In such situations, each code change bear the risk, that these corrective dependencies may be removed from the implementation space without adding a corresponding constraint inside the variability model. As a consequence, existing valid configurations may lead to invalid products in the future.

As variability information may be widely distributed in a system implementation, we need to take all artifacts into account that may contain variability information. Thus, we look at the following three spaces in our analysis:

**Configuration Space**

The configuration space is defined by the modeled variability inside the variability model. Dependencies among elements of the variability model arise through explicitly modeled constraints and through hierarchical structures, e.g., the relationship between parent and child features [MWC09].

**Implementation Space**

In product line artifacts variability is realized through variation points like preprocessor directives. Dependencies among variation points in the implementation can also be represented here. For instance, in C `#ifdef`-Blocks realizing different features may be nested inside the same code artifact, effectively creating implementation dependencies among those feature implementations.

**Build Space**

It is also possible to model dependencies among variable elements inside the build space, e.g. in `make`-scripts. Depending on the variability realization techniques used, different product line artifacts may form the final product [SVGB05, NH11]. From an analytical point of view can the complete (de-) selection of a product line artifact be treated as an `#ifdef`-Block surrounding the whole artifact. Thus, a comprehensive analysis of mismatched configuration information should also include dependencies of the build space.

In this section, we have shown how a divergence of variability of the different spaces may impair the overall product line. Such a divergence need not to lead to logical failures. Instead they may lead to an impairment of the configuration process, especially if they accumulate. So far, we described mismatched configuration information and its influence to the product line on an abstract level. In the next section, we give an example how the variability of the configuration space and the implementation space form a configuration information mismatch, leading to redundant configurations and to an erroneous code derivation.

## 3.2 Example

We introduce a small example to illustrate how missing configuration information may harm software product lines. While the variability in each space is consistent, their combination gives rise to a consistency problem.

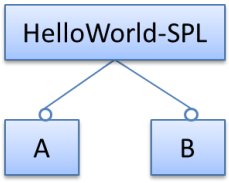



Figure 1: Feature model of our example with two optional features.

The example product line is described by the feature diagram given in Figure 1. This diagram contains two independent features. Both features are optional and are not connected by any constraints. Thus, the configuration space ($\mathcal{C}_\mathcal{S}$), defined by the feature diagram, contains four configurations: $\mathcal{C}_\mathcal{S} = \{\{\}, \{\text{A}\}, \{\text{B}\}, \{\text{A}, \text{B}\}\}$. The configuration will be saved as a set of `#define`-statements in a C-header file to `fmConfiguration.h`.

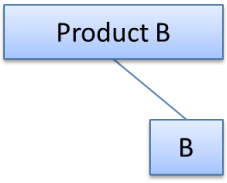
The implementation space contains only one product line artifact: `hello.c`. The variability inside this artifact is managed with preprocessor statements (cf. Listing 1). We assume in this example there is no variability contained in the build space.

Assuming the code is correct, the variability model misses configuration information. Existing analysis approaches do not aim at the detection of this mismatch. Each configuration of the configuration space $\mathcal{C}_\mathcal{S}$ will lead to a syntactically correct product and, thus, is compilable. Further, all variable parts of the implementation space (lines 6 – 13 of Listing 1) can be selected and unselected and, thus, are neither dead nor undead. However, the structure of the variability inside the configuration space differs from the structure inside the implementation space, because of the additional dependency between the two `#ifdef`-Blocks. In particular, while four different configurations exist, only three different code configurations exist. Such situations may lead to unforeseen problems (cf. Erroneous Code Derivation, Section 3.1); at least they increase unnecessary the complexity of the configuration process (cf. Redundant Configurations).

```
1  #include<stdio.h>
2  #include"fmConfiguration.h"
3
4  int main() {
5      printf("Active Features:\n");
6      #ifdef A
7      printf("Feature A\n");
8      #ifdef B
9      printf("Feature B\n");
10     #endif
11     #else
12     printf("None\n");
13     #endif
14     return 0;
15 }
```

Listing 1: Implementation space of the example.

Figure 2 shows the code after the preproccessor execution for the configuration {B}, which is identical to the configuration {}. Although the user selected Feature *B*, the product does not contain any code related to that feature (cf. Erroneous Code Derivation).




```c
#include<stdio.h>
#include"fmConfiguration.h"

int main() {
    printf("Active Features:\n");
    printf("None\n");
    return 0;
}
```

Figure 2: Effective code of the product {B}.

## 3.3 Formalization

Mismatched configuration information can be detected through a comparison of the variability model and the implemented dependencies of the (code) artifacts. In this section, we give the foundations for an efficient analysis of mismatched configuration information, based on SAT solving. In the next section, this is refined to the identification of missing nesting dependencies. The boundaries of our approach are:

**B$_1$** *Variability information can be translated into propositional logic.* The approach described below in Definition 1 relies on a satisfiability check of (extracted) information from the variability model and the assets (code artifacts, build process, ...) of the product line, to detect missing constraints. Missing information can affect the detection of missing constraints in various ways:

- The approach of Definition 1 is not applicable, if it is impossible to translate the variability information into propositional logic, even partially.
- Missing information may lead to incorrectly identified constraints (false positive results). For this reason, it is important for the translation to cover all artifacts, which may contain variability information. In the systems software domain there exist already a plethora of tools, which are capable to extract information and translate it into propositional logic.

**B$_2$** A configuration process, which makes use of a *constrained variability model* to avoid the configuration of invalid products. We have seen variability models in industry without any constraints. These are lists of configurable parameters to facilitate the configuration of all parameters at one central point. In such a case, developers need to check each new configuration manually to ensure that only valid products are derived out of the product line platform. These variability models are out of scope as we aim at the completion of existing constraints. However, our approach can be used to create an initial constrained variability model out of a lists of configurable parameters, even if it is not our scope.

We focus on software product lines with constrained variability models. For instance, public available software product lines like the Linux kernel and BusyBox make extensive use of constraints in their models [BSL+13]. These constraints are used to capture expert knowledge.

**$B_3$** A variability modeling approach, which provides *assistance to simplify the configuration process* though hiding irrelevant variables. This means, that the constraints from **$B_2$** are used to propagate values from already configured variables to dependent elements, if values become definite. For instance, in feature modeling, features become hidden or at least unselectable if their parent was deselected. Kconfig behaves similar and hides variables, whose `depends on` constraints are not fulfilled.

Our approach facilitates the usage of an SAT-solver to check whether the variability model $\mathcal{VM}$ covers a given (code) dependency $\delta$ if the aforementioned boundaries are met. Each solution of Formula 3 indicates a mismatched configuration information, which may be used to complete the variability model or for a code review:

**Definition 1.** The variability model $\mathcal{VM}$ *contains* a given (code) dependency $\delta$, if and only if $\delta$ does not further restrict the configuration space spanned by the variability model. That is the case if all valid interpretations of $\mathcal{VM}$ are also valid interpretations for $\delta$.

$$\begin{aligned} & \models \quad \mathcal{VM} \rightarrow \delta && (1) \\ \Leftrightarrow \quad & \not\models \quad \neg(\mathcal{VM} \rightarrow \delta) && (2) \\ \Leftrightarrow \quad & \not\models \quad \mathcal{VM} \wedge \neg\delta && (3) \end{aligned}$$

However, mining general dependencies $\delta_i$ from code remain an issue. The authors of [NBKC15] offer an accurate algorithm for dependency extraction. This extraction takes approximate 12 hours (with multithreading) / 386 hours (without multithreading) with two AMD Opteron processors (16 cores each) and 128 GB RAM. In this work, we focus only on a sub set, namely *missing nesting dependencies*. In the next section, we present an heuristic approach to identify those dependencies in roughly 33 minutes (without multithreading) on a much smaller hardware.

# 4 Detection of Missing Nesting Dependencies

In this section, we discuss how missing nesting dependencies can be detected efficiently by means of a SAT-based heuristic. This approach relies on the assumption that the variability information can be encoded in propositional logic. This includes the complete variability model as well as dependencies in the implementation space and the build space. For our case study, the Linux kernel, there exist already a plenty of variability extractors, which are able to translate the variability information of the different spaces into propositional logic (cf. Section 5.1). However, especially the tools for the extraction of variability information from code and build artifacts flatten the hierarchy and calculate a single condition for each variable element in the implementation and build space to simplify the downstream analysis for which they are designed for.

Our approach aims at the deduction of hierarchical dependencies $\delta$ between variable parts of the implementation and build space to check whether these hierarchies are also covered by the variability model. For instance, in a C-based implementation these hierarchical dependencies are nested preprocessor statements. Existing work combines these preprocessor conditions via a conjunction, often referred to as "presence conditions" [TLSSP11, KGR+11]. While this is sufficient to make a general statement under which conditions a certain code block is part of the resulting product, this approach does not consider hierarchies. The resulting list of presence conditions is flat, because the combination of nested conditions form a single dependency for each code block.

Our approach for extracting hierarchical dependencies $\delta$ from code and build space works in 3 steps. First, we extract conditional information from these artifacts, which serve as input data for the actual algorithm (cf. Section 5.2). Subsequent, we identify variables of the variability model, which are only used in nested conditions. We also check that the surrounding presence conditions have at least one variable in common, i.e., that there exist at least one variable which serves as a possible parent variable in a hierarchical dependency relationship. In the reminder of this report, we call this kind of relationship as a variable is *dominating* another variable. In the last step, we verify whether the nested `#ifdef`-Block can be (de-)selected if the surrounding presence condition is fulfilled.

Our approach constructs code dependencies according to the translation of feature hierarchies to propositional formula [MWC09], as code dependencies represent hierarchies in the implementation space. We translate nested conditions into an implication between the nested condition and the parent condition as illustrated in Table 1 (lines 2–3, 6–8, 11, and 13–14). In our analysis, we check each dependency in isolation. Thus, we do not need the information about the complete hierarchy as we only want to analyze the relation between a nested element and its parent elements. Therefore, we calculate the presence condition for the parent condition using the approach given in [KGR+11, TLSSP11]. This simplifies the analysis as we do not need to consider the full hierarchy in cases where the hierarchy is higher than 2 (cf. lines 4–5 and 9–10 in Table 1) as this is already checked in an earlier analysis, because each nested condition is analyzed separately.

In Section 3, we already mentioned that also the variability of the build space has to be considered for a complete analysis of mismatched configuration information. In our case study, variability in the `make`-files is used to select code files for compilation and to decide whether these files should be linked into the final product or be made loadable at runtime. These conditions can be treated like an additional `#ifdef`-Block surrounding the

Table 1: Calculation of code dependencies and dominating parent variables.

| | Code | Dependency | Dom. Vars. |
|---|---|---|---|
| 1 | `#ifdef A` | — | — |
| 2 | `#ifdef B` | $\texttt{B} \rightarrow \texttt{A}$ | A |
| 3 | $\texttt{Code}_1$ | $\texttt{B} \rightarrow \texttt{A}$ | A |
| 4 | `#ifdef C && D` | $(\texttt{C} \wedge \texttt{D}) \rightarrow (\texttt{A} \wedge \texttt{B})$ | A, B |
| 5 | $\texttt{Code}_2$ | $(\texttt{C} \wedge \texttt{D}) \rightarrow (\texttt{A} \wedge \texttt{B})$ | A, B |
| 6 | `#endif` | $\texttt{B} \rightarrow \texttt{A}$ | A |
| 7 | `#else` | $\neg\, \texttt{B} \rightarrow \texttt{A}$ | A |
| 8 | $\texttt{Code}_3$ | $\neg\, \texttt{B} \rightarrow \texttt{A}$ | A |
| 9 | `#ifdef D` | $\texttt{D} \rightarrow (\texttt{A} \wedge \neg\, \texttt{B})$ | A, B |
| 10 | $\texttt{Code}_4$ | $\texttt{D} \rightarrow (\texttt{A} \wedge \neg\, \texttt{B})$ | A, B |
| 11 | `#endif` | $\neg\, \texttt{B} \rightarrow \texttt{A}$ | A |
| 12 | `#endif` | — | — |
| 13 | `#ifdef C` | $\texttt{C} \rightarrow \texttt{A}$ | A |
| 14 | $\texttt{Code}_5$ | $\texttt{C} \rightarrow \texttt{A}$ | A |
| 15 | `#endif` | — | — |
| 16 | `#endif` | — | — |

whole code artifact. In the remainder of this report, we do not differentiate conceptually between dependencies coming from the implementation space and those coming from the build space.

These dependencies represent *local* hierarchies among variable parts of the implementation, but do not show *generalized* dependencies between variables. For instance, Table 1 shows that `B` is nested below `A` at least once, but it remains unclear whether, whenever `B` is used inside the implementation it is nested below `A`. However, these generalized dependencies are needed to verify the consistency between the variability model and the code artifacts.

We implemented a heuristic to calculate general dependencies efficiently. In a first step, we filter variables, which are dominated by other variables. For this, we calculate for each variable the intersection of all parent variables, i.e., variables used in surrounding `#ifdef`-Blocks. If a variable is used on a top level in a hierarchy, the intersection is set to the empty set. While calculating the intersection, we consider only the usage of variables of surrounding `#ifdef`-Blocks and omit sentential connectives and negations. In Table 1, `A` is not dominated, `B` and `C` are dominated only by `A`, as `C` is also used independently of `B` in lines 13 – 14, and `D` is dominated by `A` and `B`. This is done for all variables in all code artifacts. In the second step, we calculate the code dependencies for all occurrences of dominated variables as already discussed. During the later analysis, we check that the presence conditions of the surrounding `#ifdef`-Blocks are not contradictory for a variable, otherwise we do not consider them as a general dependency.

For each variable $v$, which is dominated by at least one variable permanently, we test whether its local dependencies $\delta_{v,i}$ are covered by the variability model $\mathcal{VM}$. Conceptually, this is done by testing whether $\mathcal{VM} \rightarrow \delta_{v,i}$ is a tautology (cf. Formula 1 of Definition 1), or whether $\mathcal{VM} \wedge \neg\delta_{v,i}$ (cf. Formula 3 of Definition 1) is not satisfiable. The transformation has the advantage that the variability model needs not to be transformed,

i.e., only the negation of the dependency has to be added to the existing variability model for each dependency check.

In the following section, we present our implementation for the missing nesting dependency analysis for the Linux kernel, while we evaluate our approach in Section 6.

# 5 Implementation

Here, we present our prototype for detecting missing nesting dependencies. For the implementation, we reused existing variability analysis tools to extract the relevant information from the product line artifacts, which we need for the analysis. Since the evaluation is based on the Linux kernel (cf. Section 6), only analysis tools are suitable, which can handle the variability mechanisms used for the development of the Linux kernel (Kconfig, C preprossor, and Kbuild). These tools are shown in Section 5.1. In Section 5.2, we present our algorithm to detect missing nesting dependencies.

## 5.1 Variability Extractors

In this section, we present existing variability analysis tools, which we used to extract the variability information from the Linux kernel. This information serves as input for our detection algorithm, which we explain in the next section.

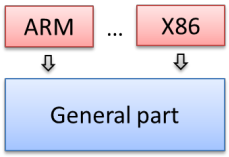



Figure 3: Simplified structure of Kconfig.

**Configuration Space**

The configuration space of the Linux kernel is managed by Kconfig [Kco15], a textual language, developed by the kernel developers. Kconfig-files consist of typed variables, which may be attributed with constraints, default values, visibility conditions, etc. [ESKS15]. Kconfig models are split in multiple files. An architecture-dependent file serves as entry point and defines architecture related settings before importing generic files, which may also be used by other architectures (cf. Figure 3).

Several research tools were developed to convert Kconfig-files into propositional logic. For our prototype, we used kconfigreader (commit from 21.11.2015)[2] to translate the Kconfig models, as this tool produces the most reliable results according to our analysis [ESKS15]. The used version contains some additional accuracy improvements, which not covered in [ESKS15].

**Implementation Space**

The Linux kernel is implemented in C, and consists of C, H, and S (assembler) files [Kbu15]. Variability is realized through preprocessor statements. Our current analysis of the implementation space takes only the C-files into account.

We use components of the Undertaker (v1.6.1)[3] tool for the extraction of the variability information from C-files. The used components parse source files and provide

[2] `https://github.com/ckaestne/kconfigreader`
[3] `https://vamos.informatik.uni-erlangen.de/trac/undertaker`

nesting structures of the preprocessor directives, as needed by our approach. We introduced a specialized main-Method for extracting the desired information without running the whole Undertaker analysis[4]. This required only minimal effort.

**Build Space**

The derivation process of the Linux kernel is implemented by Kbuild and consists of a collection of Makefiles. These files also contain variability; C-files may be excluded, linked statically into the final product, or be compiled as loadable modules depending on the configuration [NH11].

We used Makex (commit from 16.10.2014)[5] to generate presence conditions for each C-file. For this, Makex needs a list of all Linux modules; parts of Linux that can be either linked permanently, can be integrated as loadable modules, or can be excluded from the overall compilation. Each of them is represented by Kconfig as a `"tristate"` variable. We extracted this list from the output generated by kconfigreader.

## 5.2 Algorithm

Based on the principles described earlier, we outline in this section the algorithm used as a basis of our implementation. As part of our notation we use $\mathcal{V}_{KC}$ to denote all variables contained in the variability model (i.e., in Kconfig). The algorithm takes three inputs:

- the code-model ($\mathcal{CM}$) captures the variability contained in the code. It is given as a set of tuples

  **id** identifies the piece of code that creates the variability (given as file, start and end line number).

  **cond** the condition associated with the variable element; the variable element can be (for C-code) `#if`, `#else`, or `#elif`. Conditions are also associated with `else`-clauses (the negation of the corresponding `if`-clause), `elif`-clauses are handled in a similar fashion.

  **pcond** gives the complete condition of all enclosing (parent) preprocessor blocks. Conditions of multiple enclosing blocks are combined via conjunction, as also done by other variability analysis tools (cf. [TLSSP11, KGR+11]).

- the build-model ($\mathcal{BM}$), adds the dependencies given in the build-space. This input is optional, i.e., it can be replaced by an empty set $\varnothing$. The tuple-structure is intentionally identical to the code-model, although not all entries are required:

  **id** identifies the targeted source file impacted by the variability.

  **cond** describing the condition for including the target source file.

  **pcond** this is empty as no hierarchical dependency analysis exists (yet).

- the variability model ($\mathcal{VM}$), this is a CNF-representation of the variability model (Kconfig)

[4] The code is available at `https://github.com/SSE-LinuxAnalysis/pilztaker`

[5] `https://bitbucket.org/snadi/makex`

Further, we refer to individual positions in the tuples in $\mathcal{CM}$ and $\mathcal{BM}$ using the element names introduced above, we write $\mathrm{v} \in \mathit{cond}$ to test whether a variable v is contained in a formula *cond*, and we use the boolean function `sat()` to denote a satisfiability-test.

```
1  M_H = ∅
2
3  // Generate asset model
4  AM = CM ∪ BM
5
6  // Filter dominated variables v
7  for (v ∈ V_KC) do
8    // Determine parent conditions for v
9    PPC_v = {am.pcond | am ∈ AM, v ∈ am.cond}
10   // Identify dominating variables
11   P_v = {ṽ | ∀pcond ∈ PPC_v : ṽ ∈ pcond ∧ ṽ ≠ v}
12 end
13
14 // Detection of missing hierarchies
15 for (v ∈ V_KC | P_v ≠ ∅) do
16   M_v = ∅
17   for (am ∈ AM | v ∈ am.cond) do
18     if ( sat(VM ∪ {¬( am.cond → am.pcond)})
19          ∧ sat(VM ∪ {¬(¬am.cond → am.pcond)})) do
20       M_v = M_v ∪ {< am, ¬(am.cond → am.pcond) >}
21     end
22   end
23   // Check that surrounding conditions are not independent
24   if (M_v ≠ ∅ ∧ sat(¬ (⋁_{am.pcond∈PPC_v} am.pcond))) do
25     M_H = M_H ∪ {< v, M_v >}
26   end
27 end
28
29 return M_H
```

Listing 2: Algorithm for detecting missing nesting dependencies.

The output of the algorithm produces a set of identified mismatched hierarchical dependencies ($\mathcal{M}_H$), with each characterized by a tuple consisting of:

- the problematic variable v
- a set of affected code parts, related to v. Each consisting of:
  - an id to identify the code (or build-space element) that introduces the dependency *and*
  - an exclusion condition that characterizes the situation that is allowed by the variability model, but excluded by the implementation.

The algorithm starts by identifying all parent conditions related to a specific variable (line 9). It then searches for any variables that are always contained in a precondition of

a specific variable (line 11). Only those variables are of further interest for which such a "dominating" variable exists. While we are actually interested in dominating conditions, we use the variable as an indicator to identify these conditions.

Next we go through all variables that have at least one dominating variable and check their conditions (line 15/17). The satisfiability check in line 18 tests whether the negation of the code-dependency is satisfiable (and, hence, the code dependency unsatisfiable). The second satisfiability check in line 19 tests whether `am.cond` influences the code at all. This leads to a redundant configuration and, thus, points to missing configuration information, if the code dependency is indented by the developer. In line 24, we check that the combination (disjunction) of all the surrounding conditions for the identified variable v forms a restriction, i.e., there is a dependency from the nesting conditions to the surrounding conditions.

The combination of these three satisfiability checks are similar to the detection of feature effects as described in [NBKC15]. However, the algorithm presented in Listing 2 is an effective heuristic to identify hierarchical dependencies only, instead of identifying arbitrary cross-tree constraints. Please note that this does not check for dead code as in other analysis (cf. [TLSSP11]), but only identifies missing nesting dependencies, which indicate missing products/superfluous configurations as discussed in Section 3.

# 6 Evaluation

In this section, we discuss our evaluation of the approach presented in the preceding sections. First, we discuss the details of executing the analysis before we go into a discussion of the results.

## 6.1 Conducting the Analysis

As a basis for our analysis we choose the x86 architecture of Linux kernel v4.4.1. The main motivation for this choice was that so far many analysis have been performed on this product line. We took it as a particular challenge whether we could identify further problems that have not yet been described.

Our initial setup did not include an analysis of the build space, but related only the variability model with variability in the code space. After an preliminary analysis of these results we decided that we need to include build space analysis as well, otherwise the analysis returns many false positives results. An overview of the resulting setup is given in Figure 4. The preprocessing consists of three steps: (a) translating the Kconfig-information, (b) translating the makefile-information, and (c) translating the code information to propositional logic. The tools used for variability extraction were described in Section 5 as part of the implementation.

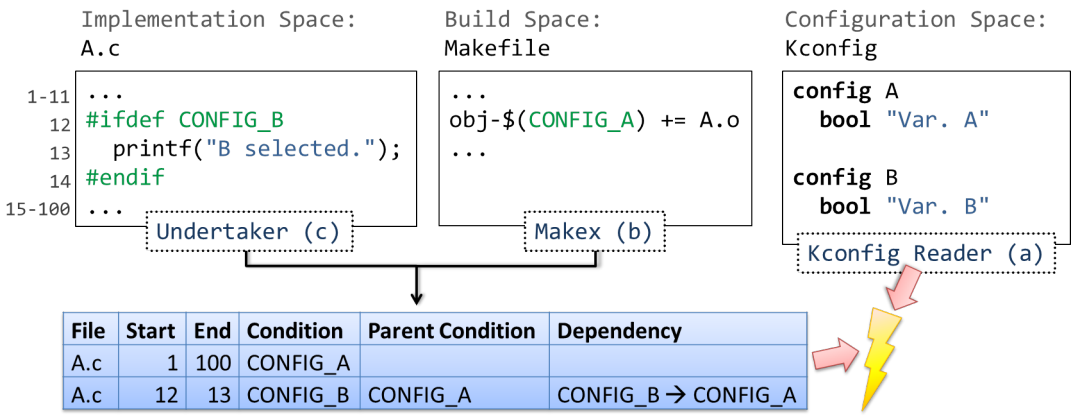



Figure 4: The analysis setup

The preprocessing steps were conducted on a computer with an Intel Core 2 6300 processor with 4 cores and 8 GB RAM in a virtualized (VMWare) Ubuntu 14.04 machine with 4GB RAM and two cores. This preprocessing took roughly 30 minutes (kconfigreader: 2:34", Makex: 0:45", and Undertaker: 26:28"). The most time-consuming part, the Undertaker run, can be reused across different architectures of the Linux kernel, as it translates all code files in an architecture-independent way. Makex and kconfigreader must be run again for each architecture.

The final analysis was performed with a Java-implementation[6] of the algorithm in Listing 2. This was executed on the host system and took 3:15 minutes. Thus, the actual analysis only requires about 10% of the total time.

[6]The code is available at `https://github.com/SSE-LinuxAnalysis/VariabilitySmellDetector`

| Variable | Result of Manual Inspection |
|---|---|
| MEMSTICK_UNSAFE_RESUME | cf. Section 6.2.1 |
| KVM_DEBUG_FS | cf. Section 6.2.2 |
| RTC_DRV_DS1685 | cf. Section 6.2.3 |
| RTC_DRV_DS1689 | |
| DEBUG_TLBFLUSH | Also used in header-files, therefore excluded from further analysis |
| DEBUG_VM_VMACACHE | |

Table 2: Results of the analysis for the x86 architecture (v4.4.1).

## 6.2 Analysis of the Results

The setup described in the previous section leads to the identification of six problem instances. Initially, we did not take into account whether the nested condition can be toggled by the user. This led to the incorrect identification of a number of Kconfig variables that were intentionally dependent on invisible variables (i.e., "HAVE_"-variables internally used in Kconfig to determine the capabilities of a certain architecture). Now, the checks in Lines 17 and 18 in Listing 2 check whether the analyzed variables can indeed be altered (or has a feature effect according to the definition of [NBKC15]). Hence, these false positives are no longer part of the result set.

While six instances that have been identified, may not seem a lot, it should be taken into account that this already includes improvements to remove false positives and that the specific Linux kernel has been intensively developed by top developers for a very long time and has been subject to a significant amount of research as well.

Manual inspection of the six results led to the following results: two elements actually had to be categorized as unclear. The reason for their identification was that the involved variables impacted the definition of macros in header files, which are included in many code files. This influenced in turn the further instantiation process. As our setup does not incorporate the analysis of header files, this was missed and led to the unclear assessment as a missing nesting dependencies.

The other four results revealed multiple code blocks associated with four variables of the Kconfig model, which led to situations in which a configuring user may need to take decisions, which are not considered during the product derivation. Without domain knowledge from an expert, i.e., a kernel developer, it remains unclear whether these detected code dependencies are accidental or intended.

### 6.2.1 MEMSTICK_UNSAFE_RESUME

Here, we want to further discuss what we consider the most problematic of these four identified results. This one appears to characterize the issue of smells in the clearest way: Our approach locates two problematic usages of the MEMSTICK_UNSAFE_RESUME-variable inside the realization of the memory stick driver. Whenever this variable is used inside the implementation space, it depends on another Kconfig-variable, namely PM. A simplified excerpt of the #ifdef-Block structure is given in Listing 3.

The comparison of the implementation space with the configuration space revealed that there is no connection between these variables in the Kconfig-model. The configuration of MEMSTICK_UNSAFE_RESUME only depends on MEMSTICK, i.e., memory sticks must

```
1    ...
2  #ifdef CONFIG_PM
3    ...
4  #ifdef CONFIG_MEMSTICK_UNSAFE_RESUME
5    ...
6  #endif /* CONFIG_MEMSTICK_UNSAFE_RESUME */
7     ...
8  #else
9    ...
10 #endif /* CONFIG_PM */
11   ...
```

Listing 3: Preprocessor structure in `mspro_block.c` (lines 1360–1435). Please note that the Linux build process adds the prefix `CONFIG_` to Kconfig variables.

be supported before their operation mode can be configured in more detail. According to its description, the variable `MEMSTICK_UNSAFE_RESUME` is used to configure the behavior of the memory stick layer when the computer is suspended. The variable `PM` can be used to enable the general functionality of suspending I/O devices.

This divergence does not allow the creation of inconsistent products, since the code of the memory stick includes the configuration of the unsafe suspension only inside the general suspension part. However, it allows situations, where a configuring user has to make decisions, which do not affect the product derivation. This is the case, if `PM` was disabled inside the "Power management and ACPI options" and the configurer wants to enable memory stick support inside the "Device Drivers" section. The relationship between these two variables remains unclear for the configurer, due to the absence of corresponding constraints in the variability model.

Based on the usage inside the implementation space and the descriptions of the variables, we suggest to model a constraint between these two variables in the Kconfig-model. Specifically, the configuration of `MEMSTICK_UNSAFE_RESUME` should be made dependent on the selection of `PM` to support the configurer during the configuration process. Further, this makes the dependencies between these variables, which are part of completely different sections inside the Kconfig-model, explicit.

We expect that this makes both code-development and maintenance of the variability-model less error-prone. Code-development would become less error-prone as a developer would no longer have to ensure through nesting of `#ifdef`-Blocks that the correct dependencies are defined, while variability modeling would become less error-prone as the relevant dependencies are explicit.

### 6.2.2 KVM_DEBUG_FS

This case is very similar to the case described in Section 6.2.1. The variable `KVM_DEBUG_FS` is only used once, in Line 628 of the file `arch/x86/kernel/kvm.c`. Through surrounding `#ifdef`-Block's and conditions of the `make`-files, this line depends of the selection of the following variables: `KVM_GUEST`, `PARAVIRT_SPINLOCKS`, and `QUEUED_SPINLOCKS`.

In Kconfig, there is only a dependency modeled between KVM_DEBUG_FS and KVM_GUEST. Consequently, the selection of KVM_DEBUG_FS is possible also when PARAVIRT_SPINLOCKS is disabled, even if this does not affect the product derivation.

All three variables are intended to affect the support of guests during the virtualization. While the first two variables are explicitly designed for the KVM hypervisor, PARAVIRT_SPINLOCKS is more general and should also affect the XEN hypervisor. There are further constraints between these variables and other virtualization related variables of the Kconfig model, but no constraints to avoid this redundant configuration.

The description of the involved Kconfig variables points out that there is a logical connection between KVM_DEBUG_FS and PARAVIRT_SPINLOCKS. For this reason, we would suggest to model a depends on constraint between these two variables. However, this relationship is not as clear as it is the case in the example of Section 6.2.1.

### 6.2.3 RTC_DRV_DS1685 and RTC_DRV_DS1689

Both variables RTC_DRV_DS1685 and RTC_DRV_DS1689 belong to RTC_DRV_DS1685_FAMILY, which adds support for the Dallas/Maxim DS1685 family of real time clocks. They are used multiple times within the Lines 866 to 1483 of the file drivers/rtc/rtc-ds1685.c and for the definition of constants in include/linux/rtc/ds1685.h. Even if the variables are also used within a header file, we decided also to include this into our manual inspection, because the usage of the defined variables could easily be traced back to the critical parts of the drivers/rtc/rtc-ds1685.c file.

Whenever RTC_DRV_DS1685 or RTC_DRV_DS1689 are used to modify the general part of the RTC_DRV_DS1685_FAMILY implementation, they are surrounded either by SYSFS or PROC_FS. Both variables are used to enable a virtual file system to provide information about the status of the operating system. However, the Kconfig model does not contain constraints between RTC_DRV_DS1685 and RTC_DRV_DS1689 to SYSFS or PROC_FS. Thus, the selection of RTC_DRV_DS1685 or RTC_DRV_DS1689 has no effect if SYSFS and PROC_FS are disabled.

Based on the documentation in Kconfig, there is no logical dependency between the involved variables. Conceptually, RTC_DRV_DS1685 and RTC_DRV_DS1689 can also be used outside of SYSFS and PROC_FS in future implementations. For this reason, we suggest not to model a constraint between these variables. However, the detected dependencies may be used for a code review to check the correctness of the identified dependencies.

# 7 Conclusion and Future Work

In this report, we introduced the problem of mismatched configuration information, which may be seen as an instance of *variability smells* as discussed by Apel et al. [ABKS13]. The main contribution, however, is a detailed analysis of a specific case: *missing nesting dependencies*. For this specific case, we present a novel and effective algorithm to detect meaningfull occurrences in a large-scale software product line. We evaluated our approach based on the x86 architecture of the Linux kernel v4.4.1. Even though the Linux kernel was already the object of many analyses, we were able to detect further issues with the approach presented in this paper. These issues constitute mismatched configuration information as the code contains variability information that cannot be identified from the variability model.

Our current analysis does not consider header-files, which led to the detection of two unclear cases. For the future, we plan to consider the consistent inclusion of header-files as discussed in [KGR+11] to increase the accuracy of our algorithm for detecting *mismatched configuration information*. We also intent to apply our analysis approach to a wider range of Linux architectures and potentially other systems.

# Acknowledgments

This work is partially supported by the Evoline project, funded by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future — Managed Software Evolution. Any opinions expressed herein are solely by the authors and not of the DFG.

# References


[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[BSL+13] Thorsten Berger, S. She, R. Lotufo, A. Wasowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[Del14] Jean Delvare. Hardware dependencies in Kconfig. Linux Mailing List `http://lkml.iu.edu/hypermail/linux/kernel/1404.1/03429.html`, 2014. Last visited 20.09.2016.

[ESKAS15] Sascha El-Sharkawy, Adam Krafczyk, Nazish Asad, and Klaus Schmid. Analysing the KConfig Semantic and Related Analysis Tools. Technical Report 1/2015, SSE 1/15/E, Institute of Computer Science, University of Hildesheim, 2015. Available at `https://sse.uni-hildesheim.de/kconfig-study`.

[ESKS15] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 45–54, 2015.

[Kbu15] Linux Kernel Makefiles. `https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt`, 2015. Last visited 03.03.2016.

[Kco15] kconfig-language. `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`, 2015. Last visited 03.03.2016.

[KGR+11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[KNO12] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, November/December 2012.

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC '13)*, pages 231–240. Carnegie Mellon University, 2009.

[NBKC14] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[NBKC15] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering*, 41(8):820–841, Aug 2015.

[NH11] Sarah Nadi and Ric Holt. Make it or break it: Mining anomalies from linux kbuild. In *18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 315–324. IEEE, 2011.

[Sch13] Klaus Schmid. A formal approach to technical debt decision making. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 153–162. ACM, 2013.

[SVGB05] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time–Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the sixth conference on Computer systems*, pages 47–60. ACM, 2011.

[vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer, 2007.

[vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation-Towards Tool Support*, page 44, 2004.