Capturing the diversity of analyses on the Linux kernel variability
Johann Mortara, Philippe Collet

To cite this version:
Johann Mortara, Philippe Collet. Capturing the diversity of analyses on the Linux kernel variability. 25th ACM International Systems and Software Product Line Conference - Volume A (SPLC ’21), Sep 2021, Leicester, United Kingdom. 10.1145/3461001.3471151. hal-03283627

HAL Id: hal-03283627
https://hal.science/hal-03283627
Submitted on 12 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Capturing the diversity of analyses on the Linux kernel variability

Johann Mortara  
Université Côte d’Azur, CNRS, I3S  
Sophia Antipolis, France  
johann.mortara@univ-cotedazur.fr

Philippe Collet  
Université Côte d’Azur, CNRS, I3S  
Sophia Antipolis, France  
philippe.collet@univ-cotedazur.fr

ABSTRACT
As its variability management architecture is complex, the Linux kernel is a constant subject of study for analyzing different aspects of its variability. It relies on a configuration-aware build system, preprocessor directives in the code, and a configuration tool. While many studies have focused on detecting anomalies within these parts or between them, all concepts and denominations are different among contributions, with similar properties devised with varied formalisms, or with no easy relationship between them. This actually hampers the understanding of all variability issues and proposed analyses, as well as their application to other highly configurable systems. In this paper, we analyse the different properties that have been studied on the variability of the kernel and propose a formalism based on the generic concepts of configurator and derivator. We instantiate them to represent the Kconfig, the Kbuild, and CPP in a unified model that enables to represent all the consistency properties. With this model, we manage to categorize the main related studies, establishing their coverage on the defined properties, showing also overlapping and divergences between studies.

CCS CONCEPTS
• Software and its engineering \(\rightarrow\) Software product lines.

KEYWORDS
Linux, configuration, variability, build system, variability anomalies

1 INTRODUCTION
With impressive figures of over 15,000 configurable features, 28 million lines of code in more than 60K files, 900,000 commits from more than 2K authors [24], the Linux kernel has been a constant subject of study for the software engineering community. Thanks to its open-source nature and its available history, software evolution [4, 16, 34] and maintenance [1, 17] issues have been extensively studied, but it is in the variability research community that it has become emblematic over the years [42].

Related work. The variability management architecture of the kernel is complex. It relies on a model-based configuration tool (Kconfig), CPP preprocessor directives in the code, and a configuration-aware build system (Kbuild).

Naturally, variability management in the Kconfig part was deeply investigated. While Sincero and Schröder-Preikschat [43] established a first mapping between the Kconfig language and feature modelling concepts, She et al. [41] investigated the inverse mapping and built a model for the Kconfig language constructs. In [40], She and Berger described the semantics of the Kconfig language, used as a basis for multiple tools [10, 19, 39, 45]. Zengler and Küchlin [53] achieved a translation of Kconfig’s constraints in a single logic formula, later reused with SAT-solving by Walch et al. [52] to analyze the consistency of Kconfig files. In his Master’s thesis, Hengelein [15] analyses defects in Kconfig. As it can be seen as a feature model [41], it can have its defects, i.e., dead features [18], or false optional features [54]. Besides, as the Linux kernel is a living ecosystem, the evolution of its variability model has also been extensively studied [9, 25, 35, 36].

Closer to the code, Sincero et al. [44] defined presence conditions to identify inconsistencies in the constraints defined by ifdef directives in the kernel, and proposed an implementation with the Undertaker toolchain. While Tartler et al. introduced the problem of inconsistencies between Kconfig files and ifdef directives in [47], they extended Undertaker to add constraints from the Kconfig files and identify inconsistencies between the two spaces [45, 46]. Other tools relying on presence conditions have been developed to reason on ifdef directives for type checking, such as TypeChef [22], and use the Linux code base as a robustness trial [20]. In [38] the authors identified more than 36,000 inconsistencies in the Linux code assets with their checking technique between FM concepts and their translation using ifdef directives.

Multiple tooled approaches have also been proposed to parse, analyze and reason on Kbuild Makefiles, as KbuildMiner [6], Makex [32], Golem [8] (and its extension MiniGolem [37]), and Kmax [13]. Berger et al. [6] analyzed the Kbuild Makefiles to extract a mapping between features and code assets in the shape of presence conditions on the features. Other tools for analyzing standard Makefiles have been applied to Kbuild files, as Makao [5], which builds a dependency graph from them. This tool is used in more recent work on the identification of unspecified dependencies in make-based systems, also applied to Kbuild [7]. Finally, after studying the internal consistency of the Kbuild Makefiles through three types of defects [30], Nadi and Holt built a third extension of Undertaker to add constraints from the Kbuild files and identify inconsistencies in the three spaces [31].

Despite this large body of work, the variability of the Linux kernel is still a subject of new studies (e.g., translating configurations from the Kconfig to propositional logic [11], using symbolic execution to recover build conditions in Kbuild files [33]), tools (e.g., interactively resolving configuration conflict [12]), and challenges (e.g., deriving a BDD [48]).

Problem statement. While many studies have focused on detecting anomalies in or between each part of the Linux build, all concepts and denominations are different among contributions, with the same properties being described with varied formalisms and sometimes different definitions, at different levels, or with no easy relationship between them.

For example, Sincero et al. [44] presented a first implementation of Undertaker for CPP implementing their formalism on CPP. Tartler et al. [45] then improved it by adding a second level
to Undertaker for reasoning over the Kconfig constraints, but the authors considered the constraints from CPP in the tool as a black box. Consequently the fine-grained comprehension of the link feature–block is then lost. The same issue can be found in the work from Nadi and Holt [31] on the Kbuild space and the third extension they provide for Undertaker.

Moreover, several studies on the variability anomalies exhibit inconsistencies in and across the different parts of the Linux build system, making use of identical denominations (such as dead and undead) for different types of code assets (code block, file,...), with sometimes colliding definitions[31, 44, 45] (cf. section 3).

These issues and the lack of a uniform vision over the different analyses on the kernel variability hamper the understanding of both the issues and the proposed solutions, as well as their transfer in the future evolution of the build system. Furthermore a uniform and consistent model could be applied to other highly configurable systems, such as MozBuild [26] from the Mozilla foundation.

Contributions. In this paper, after providing background information about variability in the Linux kernel (section 2), we tackle these issues by making the following contributions:

- We analyse the different terminologies and properties that have been studied on the variability elements of the kernel, taking as input the main related work on consistency inside Kconfig, Kbuild, and CPP, and between them (section 3).
- We bring together existing formalizations in a single formalism that captures all relevant elements of the Linux kernel variability. Instead of extracting a partial representation to reason about it, our formalism first considers selectable entities of the entire build workflow, i.e., features, files and code blocks, to express properties. These properties are determined over two concepts: a configurator, which can represent the Kconfig, and a derivator, which can be instantiated differently to represent the Kbuild (to select files) and CPP (to select code blocks) (section 4).
- We show the instantiated models and express the already identified defects from the main previous contributions, establishing their coverage on the defined properties (section 5).

Section 6 discusses the threats to the validity of this study while section 7 concludes this paper.

2 VARIABILITY IN THE LINUX KERNEL

The Linux build system is composed of three distinct stages (fig. 1):

Kconfig space. Kconfig files are present in multiple directories of the codebase and define configuration options (also called symbols) representing features. Each configuration option is defined as a config entry and can be of six different types: bool, tristate, string, hex, or int. A default value for the feature can be set with the default entry. Features can be selected directly by the user via a prompt (present in an individual prompt entry or attached to the type of the feature), or by constraints on other features (defined in a depends on entry). Menus allow to group features. If a feature is defined within a menu item that itself has a depends on entry, this condition is appended by Kconfig to the depends on condition of the feature. A feature can also force the selection of another feature with the select entry. For example, in the lib/Kconfig file presented in fig. 1, feature F00 (l.4) is a feature of type bool whose default value is y but which can be modified by the user via a prompt. To be selected, DEPS_A or DEPS_B need to be selected, and MENU_CONF needs to be satisfied. The selection of F00, will also force the selection of F_SEL. Kconfig checks for the consistency of the constraints between the selected features and outputs two files containing the list of selected features in two formats: .config will be read by the Kbuild Makefiles, and autoconf .h is a C header file that will be appended to every source file during compilation.

Make space. The Kbuild system is made of multiple Makefiles present in multiple directories throughout the project, which select objects for the compilation. Three types of objects exist: object files, directories and composite objects. Object files (such as file_c.o in lib/dir/Makefile) represent objects generated during the compilation from existing .c files in the codebase. Therefore, a file_c.c file should be present in the codebase. Added directories (such as dir/ in Makefile) will have their Kbuild Makefile parsed to select files from this subtree. Composite objects associate multiple files in one single object. For example, foo.o in lib/dir/Makefile is a composite object defined at line 2 combining file_a.o and file_b.o and used at line 1.

Selection is done by adding the object files generated at the precompilation to lists. For example, in lib/dir/Makefile, the file_c.o object is added to the obj-y list. In this case, the object will always be selected. The selection of an object can also be conditioned by the value of a feature, as for the foo.o object. CONFIG_FOO refers to the F00 feature defined in the Kconfig file lib/Kconfig. F00 is a boolean feature, therefore if it has for value y, the object will be added to the obj-y list. The same mechanism applies for the dir directory in lib/Makefile, with the small difference that BAR is a tristate feature, allowing an extra m value. The object added to the obj-m list will be compiled as a module. If a feature is not defined, the name of the list becomes obj- and is ignored.

Code space. Variability in the source files is implemented using CPP directives. Code in conditional blocks declared with #if, #elif, #ifdef, or #ifndef directives (referred to as ifdef directives) is selected only if the condition of the directive is satisfied. For example, in lib/dir/foo.c, the selection of B1 implies that the condition line 1 is true. A nested block can only be selected if its parent block is selected (the selection of B1 implies that the condition line 3 is true and that B1 is also selected). Finally, code defined in a block declared with #else or #else can only be selected if 1 if defined blocks preceding it are not selected (the selection of B3 implies that the condition line 5 is true and that B2 is not selected, and the selection of B4 implies that neither B2 nor B3 are selected).

3 DIVERSITY OF ANALYSES

Facing the many research studies on the Linux kernel variability, we queried three major digital libraries (ACM Digital Library, IEEEExplore and Scopus) searching for papers related to the consistency, build or configuration of the Linux kernel, and clearly

evoking at least one of the three mechanisms of the build system (namely, KCONFIG, KBUILD, and CPP). We then explored the references from the obtained papers to search for other work related to the topic. After manual inspection, we discarded work related to the analysis of tools to parse files from the build system [10], the tools themselves [19, 39] and tooled approaches that do not characterize anomalies [3, 6, 8, 13, 37]. We also dismiss work on the formal semantics of the KCONFIG files [40, 52, 53] and translation to boolean logic [11], which do not report anomalies. Finally, we dismiss preliminary work completed by later publications of the same authors [47] and journal extensions [32, 46].

In the following, we discuss five studies of anomalies in the kernel variability, definitions for anomalies being directly extracted from them. We mark with a * anomalies that are naturally inconsistent as they are directly extracted from papers. Characterizing these inconsistencies is done by instantiating the anomalies in our models (cf. section 5). However, some sentences may be added to reproduce the context of the definitions. For brevity’s sake, some less important definitions are not given in this paper, but are detailed in a companion report that is covering all definitions of the analysed studies, the proposed models, and their application to the kernel [28].

3.1 CPP internal consistency by Sincero et al. [44]

Sincero et al. [44] formalize CPP directives using propositional logic and propose a framework, UNDERTAKER, to automate the derivation of presence conditions from ifdef directives. They define lines of code in ifdef blocks as blocks and define for a block $b_1$ the

Presence Condition [44]:

$$PC(b_1) = \text{expression}(b_1) \land \text{noPredecessors}(b_1) \land \text{parent}(b_1)$$

with

- $\text{expression}(b_1)$: Given a block $b_1$, the function $\text{expression}(b_1)$ returns the logical expression as specified in the block declaration. Example: For the block B1 in fig. 1, the function $\text{expression}(b_1)$ returns: $A \lor B \lor C$.
- $\text{parent}(b_1)$: Given $b_1$, parent($b_1$) returns the logical variable that represents the selection of its parent. If the block is not nested in any other block, then the result is always true. Example: For the block B3 in fig. 1, the function returns: $b_1$.
- $\text{noPredecessors}(b_1)$: Given $b_1$, noPredecessors($b_1$) returns the negation of the disjunction of all its predecessors (logical variables representing blocks) in an if-group. Example: For the block B4 of fig. 1 $\text{noPredecessors}(b_4)$ returns $\neg (b_2 \lor b_3)$.

The authors then give a definition of dead defect:

**Anomaly 1 (Dead block [44])**. A block is dead if:

$$\neg \text{satisfiable}(\mathcal{K} \land C \land Block_N)$$

with $\mathcal{K}$ and $C$ the propositional formulas representing the problem space constraints (i.e., KCONFIG space) and solution space constraints (i.e., Make space) respectively. $\text{satisfiable}()$ represents the boolean satisfiability problem.

\[2\] In the remainder of this paper, we will refer to it as $\text{sat}()$. 

---

**Figure 1: Sample Linux build process, inspired from [32] and [44]**
Relying on the expression of the presence condition, the authors finally define two levels of consistency to express this definition of dead defect. As these expressions have been simplified by the authors in later work [45], they are given in the companion report.

**Anomaly 2** (Internal consistency [44]). def. in companion report.

**Anomaly 3** (External consistency [44]). def. in companion report.

3.2 **Kconfig internal consistency by Hengelein [15]**
In his Master’s thesis Hengelein [15] analyses the internal consistency of Kconfig and characterises six different types of anomalies. While the first three are common anomalies resulting from conflicts between constraints on the features, the last three are related to the syntax of the Kconfig files, and given in the companion report.

**Anomaly 4** (Dead feature [15]). A feature is dead if there are contradictions in its dependencies.

**Anomaly 5** (False optional (undead) feature * [15]). A false optional feature in Kconfig is a feature that is selected by another feature that is always on or selected by a feature that is false optional itself.

**Anomaly 6** (Missing dead feature [15]). A feature is missing dead if features in the dependencies are not defined in Kconfig.

**Anomaly 7** (Selects on Symbols with Dependencies [15]). def. in companion report.

**Anomaly 8** (Unreachable symbol [15]). def. in companion report.

**Anomaly 9** (Unnecessary Selects on Choice Values [15]). def. in companion report.

3.3 **Kbuild consistency by Nadi and Holt [30]**
Nadi and Holt [30] investigate both the internal and external consistencies of the Kbuild Makefiles by studying the (non-)use of composite objects, and the non-selection of a file because of a missing feature. The absence of files from the code base in the Makefiles is also studied (anom. [10]), but does not result from a conflict between constraints in the build system. Its definition and its instantiation in our model are thus given in the companion report.

**Anomaly 10** (File Not Used (implementation-compilation consistency) [31]). def. in companion report.

**Anomaly 11** (Feature Not Defined (compilation-configuration consistency) [31]). A .c file is referenced in the Makefile, and its presence is conditioned on a Kconfig feature being defined. However, this feature is not defined in any of the Kconfig files.

**Anomaly 12** (Variable Not Used (compilation self-consistency) [31]). A .c file is referenced in the Makefile as part of a composite variable definition, but this variable is never used.

3.4 **Kconfig–CPP consistency by Tartler et al. [45]**
Tartler et al. [45] characterize defects issuing from conflicts between the Kconfig and the CPP space. They first give the following definition of dead and undead blocks.

**Anomaly 13** (Configurability defect * [45]). A configurability defect (short: defect) is a configuration-conditional item that is either dead (never included) or undead (always included) under the precondition that its parent (enclosing item) is included:

\[
\text{dead: } \neg \text{sat}(C \land I \land \text{Block}_N) \\
\text{undead: } \neg \text{sat}(C \land I \land \neg \text{Block}_N \land \text{parent}(	ext{Block}_N))
\]

with \(C\) and \(I\) the formulas representing the configuration \((i.e., \text{Kconfig})\) and implementation \((i.e., \text{Make})\) spaces respectively.

The authors then reuse the formalism proposed by Sincero et al. [44] to simplify the defects with the two following definitions.

**Anomaly 14** (Implementation-only defects [45], simplification of anom. [2]). def. in companion report.

**Anomaly 15** (Configuration-implementation defects * [45], simplification of anom. [3]). Configuration-implementation defects occur when the rules from the configuration space contradict rules from the implementation space. We check for such defects by solving \(\text{sat}(\langle b_1 \leftrightarrow \text{PC}(b_1) \rangle \land V)\). We can infer the expressions for dead and undead configuration-implementation defects.

\[
\text{dead: } \neg \text{sat}(\langle b_1 \leftrightarrow \text{PC}(b_1) \rangle \land V) \\
\text{undead: } \neg \text{sat}(\neg (b_1 \leftrightarrow \text{PC}(b_1)) \land V)
\]

with ‘\(V\)’ the propositional formula representing the configuration space (i.e., the feature model of Kconfig).

Since anom. [14] is a special case of anom. [15] which does not consider external constraints, we only detail the latter.

**Anomaly 16** (Configuration-only defects [45]). Features are present in the configuration-space model but do not appear in any valid configuration of the model, which means that the presence condition of the feature is not satisfiable. We can check for such defects by solving: \(\text{sat}(\text{feature} \rightarrow \text{presenceCondition(feature)})\). However, no formal definition of \(\text{presenceCondition}\) was given.

**Anomaly 17** (Referential defects [45]). Referential defects are caused by a missing feature \((m)\) that appears in either the configuration or the implementation space only. That is:

\[
\text{sat}(\langle b_1 \leftrightarrow \text{PC}(b_1) \rangle \land V \land \neg (m_1 \lor \cdots \lor m_n))
\]

is unsatisfiable.

3.5 **Kconfig–Kbuild–CPP consistency by Nadi and Holt [31]**
Nadi and Holt [31] improve undertaker [45] to add constraints from the Make space and identify dead and undead artifacts at both source file and code block levels, relying on constraints from the three spaces.

**Anomaly 18** (Code anomalies [31]). def. in companion report.

**Anomaly 19** (Code–Kconfig * [31]). Code–Kconfig anomalies are defined as “Code constraints are not consistent with constraints in Kconfig” and detected using the following formulas:

\[
\text{Dead}_{\text{BN}} = \neg \text{sat}(\text{Block}_N \land C \land K) \\
\text{Undead}_{\text{BN}} = \neg \text{sat}(\neg \text{Block}_N \land \text{parent}(	ext{Block}_N) \land C \land K)
\]
These formulas are strictly identical to anom. [15], thus their expressiveness in our model will be checked together.

Anomaly 20 (Code–KCONFIG missing [31]). Such defects happen when Code constraints are not consistent with Kconfig constraints because certain features used in the code are not defined in the Kconfig files and are, therefore, always false.

Anomaly 21 (Code–Make * [31]). def. in companion report.

Anomaly 22 (Code–Make–KCONFIG * [31]). Code–Make–KCONFIG anomalies are defined as 'The combination of constraints in the three spaces are conflicting' and detected using the following formulas:

\[ \text{Dead}_{BN} = \neg \text{sat}(\text{Block}_N \land C \land M \land K) \]
\[ \text{Undead}_{BN} = \neg \text{sat}(\neg \text{Block}_N \land \text{parent}(\text{Block}_N) \land C \land M \land K) \]

Anomaly 23 (Code–Make–KCONFIG missing [31]). Such defects happen when 'The combination of constraints in the three spaces are conflicting because certain features used in the compilation constraints are not defined in the Kconfig files, and are therefore always false'.

Anomaly 24 (Make–KCONFIG * [31]). A file is dead 'if it can never be present (i.e., will never get compiled) while satisfying the combination of constraints in the Make and the KCONFIG space'. These anomalies are checked by checking these formulas.

\[ \text{Dead}_{FN} = \neg \text{sat}(\text{File}_N \land M \land K) \]
\[ \text{Undead}_{FN} = \neg \text{sat}(\neg \text{File}_N \land M \land K) \]

Anomaly 25 (Make–KCONFIG missing [31]). The definition of this type of defects is not written literally in the paper but we can derive the definition from anom. [20,23]. Such defects happen when the combination of constraints in the make and Kconfig spaces are conflicting because certain features used in the compilation constraints are not defined in the Kconfig files, and are therefore always false.

Since anom. [18,19,21] and their missing variants anom. [20,25] encompass special cases of anom. [19] and anom. [23] by considering only some spaces, we only detail the instantiation of the latter.

3.6 Summary

While the studies on the Linux build system all present the three spaces (KCONFIG, KBUILD and CPP) when they used them, no standard denomination is given to each of these spaces, leading to a profusion of names. The different denominations for the studied spaces are conflicting because certain features used in the compilation constraints are not defined, then Block 2 is not undead according to anom. [15] as it is not always included.

While encompassing the variability of the whole Linux kernel build system, the models presented in the next section will also help in obtaining a coherent set of consistency properties.

4 PROPOSED MODELS

In order to cover all three stages of the Linux kernel build system and to represent its variability mechanisms independently of their implementation, we design a model with the concepts of feature, asset, propositional formula, and presence condition. We add a concept of predecessor to handle the dependency between the three stages, as well as two additional concepts to be able to represent the stages themselves:

- a configurator defines presence conditions on features (i.e., the condition allowing an individual feature to be selected). Presence conditions on features are propositional formulas on other features;
- a derivator defines presence conditions on assets, (i.e., the condition allowing an individual asset to be selected). Presence conditions on assets are propositional formulas on both features and assets, which can be either of the same type or of another type.

Some properties will also be defined on the internal and external consistency of the elements as to cover the different anomalies devised in the previous section. We could also have built our models

<table>
<thead>
<tr>
<th>Paper</th>
<th>KCONFIG files</th>
<th>KBUILD Makefiles</th>
<th>CPP / Source files</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>Kconfig space</td>
<td>Make space</td>
<td>Code space</td>
</tr>
<tr>
<td>[15]</td>
<td>Feature Modeling Configuration</td>
<td>Build system</td>
<td>Generator Preprocessor</td>
</tr>
<tr>
<td>[30]</td>
<td>Configuration space</td>
<td>Compilation space</td>
<td>Implementation space</td>
</tr>
<tr>
<td>[44]</td>
<td>Problem space Model</td>
<td>/</td>
<td>Solution space Implementation</td>
</tr>
<tr>
<td>[45]</td>
<td>Configuration space</td>
<td>/</td>
<td>Implementation space</td>
</tr>
</tbody>
</table>

Details of all the studies presented in the introduction are given in a complete table in the companion report [28].

Block 2 is undead according to anom. [13], as the selection of its parent (Block 1) implies its selection. However, if the A variable is not defined, then Block 2 is not undead according to anom. [15] as it is not always included.

The combination of constraints in the three spaces are conflicting.
on a more expressive theoretical background, such as the refinement theory, to potentially obtain for free some properties, but we decided to rely on a more simple but very explicit basis to clarify first all concepts and inconsistencies.

In the following, we will use these utility definitions:

- $\text{terms}(\phi)$ a helper function which, given a propositional formula, returns the terms in it (e.g., $\text{terms}(A \land B \lor C) = \{A, B, C\}$).
- $\text{expand}(\phi)$ a helper function which, given a propositional formula $\phi$, replaces every asset $a$ in $\phi$ by its presence condition, noted $PC_{\text{Int}}(a)$ and defined in def. 2 (e.g., $\text{expand}(b_1 \land \neg b_2) = PC_{\text{Int}}(b_1) \land \neg PC_{\text{Int}}(b_2)$).
- $\text{slice}(C, T)$ an operator which, given a set of boolean conditions on terms $C$ and a set of terms $T$, returns the conjunction of all propositional formulas from $C$ containing terms from $T$. The operator is recursively applied to the terms that appear in these formulas.

### 4.1 Derivator Model

In this section, we introduce the concepts to form the derivator model and illustrate them with its application to CPP.

#### Definition 1 (Asset).

An asset $a$ is $\langle \phi_{\text{select}}, \phi_{\text{preds}}, \phi_{\text{depInt}}, \phi_{\text{depExt}} \rangle$ from a set of assets $\mathcal{A}$ is defined as follows:

- $\phi_{\text{select}}$ is a propositional formula for the asset’s selection;
- $\phi_{\text{preds}}$ is a propositional formula on other assets that are evaluated before $a$. We call these assets predecessors;
- $\phi_{\text{depInt}}$ is a propositional formula on assets on which $a$ is dependent;
- $\phi_{\text{depExt}}$ is a propositional formula on assets from another context on which $a$ is dependent.

#### Application to CPP.

An asset $b$ is a code block, with:

- $\phi_{\text{select}}$ the condition of the #if surrounding the block;
- $\phi_{\text{preds}} = \neg (\lor_i b_i)$ if $b$ is an #elsi or #else block, $b_i$ represents the corresponding #if block and the potential #elsi blocks before $b$;
- $\phi_{\text{depInt}} = p$ with $p$ the parent block of $b$ if $b$ is a nested block.
- $\phi_{\text{depExt}} = f$ if the file containing $b$.

#### Example.

In fig. 1, the lib/foo/foo.c file and the blocks B1, B2, B3, and B4 it contains are represented by the following assets:

- $\text{file} = (\text{true}, \text{true}, \text{true})$
- $b_1 = (\text{FOO} \lor \text{BAR} \lor \text{BAZ}, \text{true}, \text{true}, \text{file})$
- $b_2 = (\text{FOO}, \text{true}, b_1, \text{file})$
- $b_3 = (\text{BAR}, \neg b_2, b_1, \text{file})$
- $b_4 = (\text{true}, \neg (b_2 \lor b_3), b_1, \text{file})$

#### Definition 2 (Internal presence condition).

The internal presence condition of an asset is the boolean formula that needs to be satisfiable for the asset to be selectable. It is defined as

$$PC_{\text{Int}}(a) = \phi_{\text{select}} \land \text{expand}(\phi_{\text{preds}}) \land \text{expand}(\phi_{\text{depInt}})$$

#### Note.

An asset is selected if and only if its presence condition is satisfied: $PC_{\text{Int}}(a) \iff a$

#### Application to CPP.

Let us take again the previous example.

$$PC_{\text{Int}}(b_1) = \phi_{\text{select}} \land \text{expand}(\phi_{\text{preds}}) \land \text{expand}(\phi_{\text{depInt}})$$

#### Application to CPP.

In the Linux build system, the selection of a CPP block is conditioned by constraints on both the features used in the #if instructions (which are determined at the KCONFIG level) and the file containing the block (which are determined at the KBUILD level). Thus, the context $C$ to express the external presence condition of a block is the union of the KCONFIG and KBUILD contexts $C = C_{\text{KCONFIG}} \cup C_{\text{KBUILD}}$.

Let us take an example with:

- $C_{\text{KCONFIG}} = \{\text{FOO} \lor \text{BAR} \lor \text{BAZ} \lor (\neg F1), F1 \lor (\neg \text{FOO}), F3 \lor F4\}$
- $C_{\text{KBUILD}} = \{\text{file} \leftrightarrow \text{FOO}\}$

Then $PC_{\text{Ext}}(b_1) = PC_{\text{Int}}(b_1) \land \text{slice}(C, \text{terms}(PC_{\text{Int}}(b_1)) \cup \text{terms}(\phi_{\text{depExt}}))$

#### 4.1.1 Internal consistency.

To express defects, we define dead, core, and full-mandatory assets, relying on definitions of dead and false-option features introduced by Benavides et al. [5], and full-mandatory features from Trinidad et al. [49].

#### Definition 4 (Dead asset).

An asset $a$ of $\mathcal{A}$ is dead if it can never be selected. The set of dead assets of $\mathcal{A}$ is noted $\text{deads}(\mathcal{A})$.

$$a \in \text{deads}(\mathcal{A}) \iff \neg \text{sat}(PC_{\text{Int}}(a))$$
**Table 2: Truth table for \( \phi_{\text{enable}} \) from a KCONFIG feature**

<table>
<thead>
<tr>
<th>Presence of prompt</th>
<th>Presence of default</th>
<th>( \phi_{\text{enable}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>activated</td>
<td>true</td>
</tr>
<tr>
<td>no</td>
<td>not activated</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

In the KCONFIG file presented in fig. 1, three features are defined: \( \text{FOO} \), \( \text{BAR} \), and \( \text{F_SEL} \). Existing work on the semantics of the KCONFIG files [40] inline the conditions from the menu items surrounding the definition of a feature in the depends on condition. These features can be represented by the following cases:

- \( \text{FOO} = \{ \text{true}, (\text{DEPS}_A \lor \text{DEPS}_B) \land \text{MENU_COND}, \{ \} \} \)
- \( \text{BAR} = \{ \text{true}, \text{MENU_COND}, \{ \} \} \)
- \( \text{F_SEL} = \{ \text{false}, \text{true}, \{ \text{FOO} \} \} \)

**Definition 10** (Presence condition). The presence condition of a feature \( F \in \mathcal{F} \) represents the boolean formula which needs to be satisfied for the feature to be selected.

\[
\mathcal{P}C(F) = (\phi_{\text{enable}} \land \text{expand}(\phi_{\text{deps}})) \lor \text{directSelect}(F)
\]

with \( \text{directSelect}(F) = \bigvee_{F_i \in \text{select}} \mathcal{P}C(F_i) \).

**Note.** The selection of a feature implies that its presence condition is satisfied: \( F \rightarrow \mathcal{P}C(F) \) There is no bimilation as we consider that a user can manually interfere in the selection. Therefore, the information extracted from the model can only express if a feature can be selected, and not its effective selection.

**Application to KCONFIG.**

\[
\begin{align*}
\mathcal{P}C(\text{FOO}) &= (\phi_{\text{enable}}(\text{FOO}) \land \text{expand}(\phi_{\text{FOO}})) \lor \text{directSelect}(\text{FOO}) \\
&= \text{true} \land ((\mathcal{P}C(\text{DEPS}_A) \lor \mathcal{P}C(\text{DEPS}_B)) \land \mathcal{P}(\text{MENU_COND})) \\
&= (\mathcal{P}C(\text{DEPS}_A) \lor \mathcal{P}C(\text{DEPS}_B)) \land \mathcal{P}(\text{MENU_COND}) \\
\mathcal{P}C(\text{BAR}) &= (\phi_{\text{enable}}(\text{BAR}) \land \text{expand}(\phi_{\text{BAR}})) \lor \text{directSelect}(\text{BAR}) \\
&= \text{true} \land \mathcal{P}(\text{MENU_COND}) \\
&= \mathcal{P}(\text{MENU_COND}) \\
\mathcal{P}C(\text{F_SEL}) &= (\text{false} \land \text{true}) \lor \mathcal{P}C(\text{FOO}) = \mathcal{P}C(\text{FOO})
\end{align*}
\]

**Note.** Due to the size and complexity of the KCONFIG model, obtaining a sound and complete abstraction of its semantics is still a challenge. The latest studies on boolean translation are not able to represent the whole complexity of the language [11]. Because of these limitations, the accuracy of variability reasoning approaches is also limited and acknowledged by researchers [12]. Therefore, we aim here to provide a model allowing us to synthesize the current work, and do not pretend to present a complete model of KCONFIG.  

### 4.2 Configurator Model

The configurator represents the model element that checks the selection of features. It is represented by a set of features \( \mathcal{F} \). We will illustrate the formalization here with its application to the KCONFIG.

\[
\begin{align*}
F &= (\phi_{\text{enable}}, \phi_{\text{deps}}, \mathcal{F}_{\text{select}}) \\
&= (\text{enable propositional formula representing the ability to select feature}) \\
&= (\text{deps propositional formula on features on which } F \text{ dependent}) \\
&= (\text{select boolean formula on features defined in the depends on statement}) \\
&= (\text{set of features automatically selecting } F \text{ if a feature from } \mathcal{F}_{\text{select}} \text{ is selected, } F \text{ also selected, regardless of the precedent conditions.})
\end{align*}
\]

**Application to KCONFIG.** A feature \( F \) is a configuration option defined in a KCONFIG file, with:

- \( \phi_{\text{enable}} \) represents the ability to select the feature by user selection (prompt), or default value, as defined in table 2.
- \( \phi_{\text{deps}} \) represents the boolean formula on features defined in the depends on statement.
- \( \mathcal{F}_{\text{select}} \) is a set of features selecting \( F \) with a select statement.

**Definition 11** (Dead feature). A feature \( F \in \mathcal{F} \) is dead if it can never be selected. The set of dead features is noted \( \text{deadFeatures}() \).

\[
F \in \text{deadFeatures}() \iff \neg \text{select}(\mathcal{P}C(F))
\]

---

**Note.** This consistency check includes the more specific case where an asset is dead because of an inconsistency with the condition to select its internal dependencies (i.e., \( \text{expand}(\phi_{\text{depsIni}}) \rightarrow \neg \text{select} \)) as in this case \( \mathcal{P}C(\text{Int}(a)) \) is inconsistent.

**Definition 5** (Core asset). An asset \( a \in \mathcal{A} \) is a core asset if it is always selected. The set of core assets of \( \mathcal{A} \) is noted \( \text{core}(\mathcal{A}) \).

\[
a \in \text{core}(\mathcal{A}) \iff \neg \text{select}(\neg \mathcal{P}C(\text{Int}(a)))
\]

### 4.2 External consistency.

**Definition 6.** (Externally dead asset) An asset \( a \) is an externally dead asset if it is never selected due to inconsistencies with its context. The set of externally dead assets of \( \mathcal{A} \) is noted \( \text{deadExt}(\mathcal{A}) \).

\[
a \in \text{deadExt}(\mathcal{A}) \iff \neg \text{select}(\neg \mathcal{P}C(\text{Ext}(a)))
\]

**Definition 7** (Externally core asset). An asset \( a \in \mathcal{A} \) is an externally core asset if it is always selected independently of the constraints of the context. The set of core assets of \( \mathcal{A} \) is noted \( \text{coreExt}(\mathcal{A}) \).

\[
a \in \text{coreExt}(\mathcal{A}) \iff \neg \text{select}(\neg \mathcal{P}C(\text{Ext}(a)))
\]

**Definition 8.** (Externally full-mandatory asset). An asset \( a \) is an externally full-mandatory asset if the selection of its parent dependencies implies its selection due to the formulas in its context. The set of externally full-mandatory assets of \( \mathcal{A} \) is noted \( \text{fullExt}(\mathcal{A}) \).

\[
a \in \text{fullExt}(\mathcal{A}) \iff \text{select} \left( \phi_{\text{deps}}(\mathcal{A})(\text{Ext}(a)) \right)
\]

**Definition 9.** (Missing dead asset). An asset \( a \) is missing dead if a feature in its presence condition is not defined in the context \( C \). The set of missing assets is noted \( \text{missing}(\mathcal{A}) \).

\[
a \in \text{missing}(\mathcal{A}) \iff \exists m \in \text{terms}(\mathcal{P}C(\text{Ext}(a))) \text{ if } (m \notin \text{terms}(C))
\]

---

Capturing the diversity of analyses on the Linux kernel variability
We now instantiate our model on the kernel build system. The con-

Note. If \( F_S \in \mathcal{F}_{select} \) is a core feature, then \( F \) is also a core feature, as \( \mathcal{PC}(F_S) \rightarrow \mathcal{PC}(F) \).

Definition 13. (Missing dead feature) A feature \( F \) is missing dead if a feature in its presence condition is not defined. The set of missing dead features is noted \( \text{missingDeadFeatures()} \).

\[
F \in \text{missingDeadFeatures()} \iff (m \in \text{terms}(\mathcal{PC}(F)) \land (m \notin \mathcal{F})
\]

5 INSTANCIATION ON THE LINUX KERNEL

We now instantiate our model on the kernel build system. The con-

\[
\text{Expression for its related anomalies.}
\]

5.1 Model on CPP

For CPP we have to describe the presence conditions and cross-

Compliance with presence conditions from Sincero et al. [44]. For conciseness and to prevent confusion, we name this definition \( \mathcal{PC}_{Sin} \) and use the more compact expression given in [45]:

\[
\mathcal{PC}_{Sin}(b) = \text{expr}(b) \land \text{noPredecessors}(b) \land \text{parent}(b)
\]

We can express \( \mathcal{PC}_{Sin} \) using our definition of asset from def. 1. Let us apply \( \mathcal{PC}_{Sin} \) on an asset \( b \) as defined in section 4.1.

\[
\mathcal{PC}_{Sin}(b) = \text{expr}(b) \land \text{noPredecessors}(b) \land \text{parent}(b)
\]

\[
= \phi_{selects} \land (\neg \text{pred}_1 \lor \neg \text{pred}_2 \lor \ldots \lor \neg \text{pred}_n) \land \phi_{depInt_b}
\]

\[
= \phi_{selects} \land \phi_{preds_b} \land \phi_{depInt_b}
\]

\( \phi_{preds_b} \) and \( \phi_{depInt_b} \) are propositions on assets corresponding to the blocks themselves. However, to evaluate the presence condition, these assets have to be expanded to their logical expression.

\[
\mathcal{PC}_{Sin}(b) = \phi_{selects} \land \text{expand}(\phi_{preds_b}) \land \text{expand}(\phi_{depInt_b})
\]

The definition of \( \mathcal{PC}_{Sin} \) is therefore compliant with our definition of \( \mathcal{PC}_{Int} \) given in def. 2.

5.1.2 Expressing cross-space formulas. Nadi and Holt [31] defined multiple anomalies (anom. [19,21,22,24]) using different terms, i.e., \( B_N, C, M, \) and \( K \), which we now describe with our model.

\( B_N \land C. B_N \) represents a block, and \( C \) the constraints in the code space. This expression is true if and only if the block \( B_N \) is selected, thus it corresponds to \( B_N \leftrightarrow \mathcal{PC}_{Sin}(B_N) \) using Tartler et al. [45]’s notation and \( \mathcal{PC}_{Int}(B_N) \) in our model.

\( \text{parent}(B_N). \text{parent}(B_N) \) represents the selection of the parent of a block, i.e., its enclosing block. This expression corresponds to \( \text{expand}(\phi_{depInt_{B_N}}) \) in our model.

The \( \mathcal{Kconfig} \) space. \( K \) represents the set of constraints in the \( \mathcal{Kconfig} \) space, i.e., the constraints on features that allow them to be selected. Tartler et al. [45] do not use the whole feature model expression as the solving would not scale. They instead identify the features impacting the selection of a given code block using a slicing algorithm to build a minimal but sufficient subset of the configuration space through a recursive application on each new feature found in the presence condition expression.

The make space. \( M. M \) represents the set of constraints in the make space, i.e., the constraints on features that allow the selection of source files in the Makefiles. In her PhD thesis [29], Nadi states: since the conflicts in anom. [21] arise from looking at the block presence condition as well as the file’s presence condition, we call this category of anomalies code-build anomalies. Thus, to detect defects involving the make space, it is only necessary to have the presence condition of the file containing the analyzed block.

In section 4.1, we instantiate on CPP the definition of external presence condition given in def. 3, using for context \( C = \mathcal{Kconfig} \cup \mathcal{Kbuild} \). Thus, \( \mathcal{Kconfig} = K \) and \( \mathcal{Kbuild} = M \), and:

\[
slice(C_{\mathcal{Kconfig}}, \text{terms}(\mathcal{PC}_{\mathcal{Int}}(a)) \cup \text{terms}(\phi_{\text{depExt}})) \models K
\]

\[
slice(C_{\mathcal{Kconfig}}, \text{terms}(\mathcal{PC}_{\mathcal{Int}}(a)) \cup \text{terms}(\phi_{\text{depExt}})) \models M
\]

We can then express the different formulas in our model.

Instantiation 1 (Expressing code–make–\( \mathcal{Kconfig} \) anomalies [22]).

\[
\neg \text{sat}(B_N \land C \land M \land K)
\]

\[
\iff \neg \text{sat}(\mathcal{PC}_{\mathcal{Int}}(B_N) \land C)
\]

\[
\iff \neg \text{sat}(\mathcal{PC}_{\mathcal{Ext}}(B_N)) \text{ (def. 3)}
\]

\[
\neg \text{sat}(\neg B_N \land \text{parent}(B_N) \land C \land M \land K)
\]

\[
\iff \neg \text{sat}(C \land \neg \mathcal{PC}_{\mathcal{Int}}(B_N) \land \text{expand}(\phi_{\text{depInt}_{B_N}}))
\]

\[
\iff \neg \text{sat}(\neg \mathcal{PC}_{\mathcal{Ext}}(B_N) \land \text{expand}(\phi_{\text{depInt}_{B_N}}))
\]

anom. [22] thus expresses dead (def. 6) and full-mandatory defects (def. 8).

Instantiation 2 (Expressing configurability defects [13]). Same as inst. 1 with \( C = \mathcal{Kconfig} \).

Instantiation 3 (Expressing code–make–\( \mathcal{Kconfig} \) [19]). Same as inst. 2.

Instantiation 4 (Expressing configuration-implementation defects [15]). \( V \) corresponds to the minimum but sufficient set of constraints from the configuration space. Thus:

\[
\text{sat}(b_1 \leftrightarrow \mathcal{PC}(b_1)) \land V \iff \text{sat}(\mathcal{PC}_{\mathcal{Int}}(b_1) \land \mathcal{Kconfig})
\]

We can then express \textit{dead} and \textit{undead} configuration-implementation defects. Given \( B \) the set of blocks and \( C = \mathcal{Kconfig} \):

\[
\neg \text{sat}(b_1 \leftrightarrow \mathcal{PC}(b_1)) \land V \iff \neg \text{sat}(\mathcal{PC}_{\mathcal{Int}}(b_1) \land C)
\]

\[
\iff \neg \text{sat}(\mathcal{PC}_{\mathcal{Ext}}(b_1)) \text{ (def. 3)}
\]

\[
\neg \text{sat}(\neg (b_1 \leftrightarrow \mathcal{PC}(b_1)) \land V) \iff \neg \text{sat}(\neg \mathcal{PC}_{\mathcal{Int}}(b_1) \land C)
\]

\[
\iff \neg \text{sat}(\neg \mathcal{PC}_{\mathcal{Ext}}(b_1)) \text{ (def. 3)}
\]

anom. [15] thus expresses dead (def. 6) and core defects (def. 7).
Instantiation 5 (Expressing code–Make–KCONFIG missing defects [23]). We showed in inst. 1 that $B_N \land C \land M \land K \models PC_{Ext}(B_N)$. If a feature $m$ from the formula is not defined in the KCONFIG files, it means that $m \notin \text{terms}(K) \cup \text{terms}(M)$, i.e., $m \notin \text{terms}(C_{KCONFIG} \cup C_{KBUILD})$. Therefore, $\exists m \in \text{terms}(PC_{Ext}(B_N)) \mid (m \notin \text{terms}(C))$, thus $B_N$ is dead by missing feature.

Instantiation 6 (Expressing code–KCONFIG missing defects [20]). Same as inst. 5 with $C = C_{KCONFIG}$.

Instantiation 7 (Expressing referential defects [17]). If the feature is missing in the configuration space, then the definition corresponds def. 9 with $C = C_{KCONFIG}$ as context. A feature missing in the implementation space can mean that the feature is used in the Make space only. It is characterized as a defect as [45] does not consider this space, but it is not a defect for us.

5.2 Model on KBUILD

At the KBUILD level, an asset $s = (\phi_{\text{select}}, \phi_{\text{preds}}, \phi_{\text{depInt}}, \phi_{\text{depExt}})$ can represent a $C$ object file. We then express presence conditions and related anomalies with our model. As seen in section 2, an object is selected for compilation by being added to defined lists, with possible constraints on one or more features in case of multiple definitions. Before, objects can also be added to composite variables.

- $\phi_{\text{select}} = \bigvee f_i$ with $f_i$ being features which at least one needs to be set for the source file to be selected. If the asset is always selected, $\phi_{\text{select}} = \text{true}$.
- The asset is defined but never added to a list, $\phi_{\text{select}} = \text{false}$.
- $\phi_{\text{preds}} = \text{comp}$ with $\text{comp}$ the name of the composite variable if $s$ is part of a composite definition. $\text{comp}$ must be selected.
- $\phi_{\text{depInt}} = \text{dir}$ with $\text{dir}$ the directory containing the source file represented by $s$ which also needs to be selected.
- $\phi_{\text{depExt}} = \text{true}$ as the selection of a source file only relies on its feature.

Expressing the assets from fig. 1.

- $\text{dir} = (\text{BAR}, \text{true}, \text{true}, \text{true})$
- $\text{foo} = (\text{FOO}, \text{false}, \text{true}, \text{true})$
- $\text{file_a} = (\text{true}, \text{true}, \text{true})$
- $\text{file_b} = (\text{true}, \text{true}, \text{true})$
- $\text{file_c} = (\text{true}, \text{true}, \text{true})$

Expressing their presence conditions

- $PC_{\text{Int}}(\text{dir}) = \phi_{\text{select}_{\text{dir}}} \land \text{expand}(\phi_{\text{preds}_{\text{dir}}}) \land \text{expand}(\phi_{\text{depInt}_{\text{dir}}})$
  
  $= \text{BAR} \land \text{true} \land \text{true} = \text{BAR}$

- $PC_{\text{Int}}(\text{foo}) = \text{FOO} \land \text{true} \land \text{false} = \text{FOO} \land \text{BAR}$

- $PC_{\text{Int}}(\text{file_a}) = \text{true} \land \text{false} = \text{true}$

- $PC_{\text{Int}}(\text{file_b}) = \text{true} \land \text{false} = \text{true}$

- $PC_{\text{Int}}(\text{file_c}) = \text{true} \land \text{false} = \text{true}$

Instantiation 8 (Expressing Feature Not Defined [11]). Given $m$ a feature not being defined in any KCONFIG files, and $a$ a file referenced a KBUILD Makefile whose presence is conditioned by $m$. Thus, $m$ is present in $\phi_{\text{select}_a}$, however is not present in the features defined in the KCONFIG files, obtained with $\text{terms}(C_{KCONFIG})$.

- $(m \in \text{terms}(\phi_{\text{select}_a})) \land (m \notin \text{terms}(C_{KCONFIG}))$

As terms($\phi_{\text{select}_a}$) $\subseteq$ terms($PC_{Ext}(a)$), anom. [11] is a special case of def. 9, therefore $a$ is a missing dead file.

Instantiation 9 (Expressing Variable Not Used [12]). Given $a$ an asset and $\phi_{\text{preds}_a} = \text{comp}$. $a$ is an unused variable if $\neg\phi_{\text{select}_{\text{comp}}}$ and $\neg\phi_{\text{select}_{\text{comp}}} \rightarrow 

\neg PC_{\text{Int}}(\text{comp}) \rightarrow \neg \text{expand}(\phi_{\text{preds}_a}) \rightarrow \neg PC(a)$. Thus, $a$ is a dead asset.

Instantiation 10 (Expressing Make–KCONFIG anomalies [24]). Let us consider $s$ the asset that represents the file $F_N$, and $C = C_{KCONFIG}$.

- $PC_{\text{Int}}(s) = \phi_{\text{select}} \land \text{expand}(\phi_{\text{preds}}) \land \text{expand}(\phi_{\text{depInt}})$
  
  $= \phi_{\text{select}} \land PC_{\text{Int}}(\text{comp}) \land PC_{\text{Int}}(\text{dir})$

To build $M$ as it appears in anom. [24], Nadi and Holt [31] extract for every file a presence condition consisting of a conjunction of the features conditioning the selection of the file ($\bigvee f_i = \phi_{\text{select}_i}$), the composite object if present ($PC_{\text{Int}}(\text{comp})$) and its parent directory ($PC_{\text{Int}}(\text{dir})$) in the corresponding Makefiles. Therefore, $PC_{\text{Int}}(s) \models (F_N \land M)$.

- $PC_{\text{Ext}}(s) = PC_{\text{Int}}(s) \land \text{slice}(C, \text{terms}(PC_{\text{Int}}(s))) \cup \text{terms}(\phi_{\text{depExt}})$
  
  $\models (F_N \land M) \land K$ (cf. section 5.1.2)

We can then express anom. [24] in our model:

- $\neg \text{sat}(F_N \land M \land K) \iff \neg \text{sat}(PC_{\text{Ext}}(s))$
- $\neg \text{sat}(\neg F_N \land M \land K) \iff \neg \text{sat}(\neg PC_{\text{Ext}}(s))$

anom. [24] thus expresses dead (def. 6) and core defects (def. 7).

Instantiation 11 (Expressing Make–KCONFIG missing defects [25]). Same as inst. 5 with $C = C_{KCONFIG}$, and relying on formulas from inst. 10.

5.3 Model on KCONFIG

Given the configurator model and the application example already given in section 4.2, we just have to instantiate the anomalies.

Instantiation 12 (Expressing dead feature [4]). Given $F$ a dead feature. The definition can be expressed in our model as $\neg \text{sat}(\phi_{\text{dep}})$, which itself implies $\neg \text{sat}(PC(F))$, hence $F$ is dead.

Instantiation 13 (Expressing false optional [5]). This definition corresponds to the note in def. 12, thus $F$ is a core feature.

Instantiation 14 (Expressing missing dead feature [6]). The definition limits the presence of an undefined feature in the dependencies:

- $(m \in \text{terms}(\phi_{\text{dep}})) \land (m \notin F)$

As terms($\phi_{\text{dep}}$) $\subseteq$ terms($PC(F)$), every missing dead feature according to anom. [6] is also missing dead in our model.

Instantiation 15 (Expressing configuration-only defects [16]). The function $\text{presenceCondition(feature)}$ returns the presence implication of the feature and is defined by the authors as "the selection of the feature itself and the expression of the depends on option." This definition, expressed by our model, corresponds to $\phi_{\text{enable}_f} \land \text{expand}(\phi_{\text{dep}}) = PC(f)$. Thus

- $\neg \text{sat}(f \rightarrow \text{presenceCondition}(f)) \iff \neg \text{sat}(PC(f))$

Therefore, $f$ is dead.
5.4 Resulting coverage

From the instantiations of the configurator model on \texttt{Kconfig} and the derivator on both \texttt{Kbuild} and \texttt{CPP}, we obtain a complete expression of the different formulas and anomalies taken as input. A summary of the different anomalies for each paper and how they are expressed is presented in table 3. As expected, no existing proposal expresses defects in every space of the Linux build system. The table confirms the inconsistencies that we manually observed in section 3.6 between anom. {13} and anom. {15} from [45], with the first anomaly being characterized as a \textit{full-mandatory} defect and the other as a \textit{core} defect. Moreover, similar inconsistencies are exhibited in defects from [31], as well as two anomalies that are described as \textit{dead} defects but are not called as such (anom. {12,16}).

6 THREATS TO VALIDITY

\textit{Internal threats to validity.} A first internal threat could be caused by the selection of papers made to devise the properties and the model. However, given the narrow focus of the subject and the fact that we sought additional work in the references of the obtained papers, we believe that the most important studies are included.

Another internal threat concerns the accuracy of the formalisms from the chosen studies, but their application to the kernel was demonstrated. Besides, while we provide a formalization, there is no proof of correctness of the formalism nor associated syntax and semantics given in a theorem prover. As future work, we plan to rather use modelling tools to extract representations of the different spaces in our model and reason on properties.

\textit{External threats to validity.} While the application of the proposed models to the Linux build system is shown in the paper, there is no demonstration of the applicability of the configurator and derivator concepts to other build systems. A first demonstration of the derivator concept is nevertheless done through its double instantiation as the \texttt{Kconfig} and \texttt{CPP} preprocessor. Our future plan also includes building a generalized framework and applying it to other variability build systems [14, 21, 26, 27].

7 CONCLUSION

The Linux kernel is a constant subject of study for the software variability research community. While many aspects of its build system have been studied, no uniform model encompassing all properties on the \texttt{Kconfig}, \texttt{Kbuild}, and \texttt{CPP}, has been proposed.

In this paper, we first examined the main studies on the kernel variability to propose a complete set of definitions for the analyzed properties. We exposed the differences in terminology and some inconsistencies in the interpretation of similar definitions in them. We then described a formalism based on the generic concepts of configurator and derivator to express the whole set of consistency properties. We showed that the configurator can be instantiated to represent the \texttt{Kconfig}, while the instantiated derivators can represent the \texttt{Kbuild}, selecting files, or \texttt{CPP}, selecting code blocks. The obtained model enables one to categorize the previous studies and to establish their coverage and divergences on the analyses.

As future work, we plan to implement a model-driven framework derived from the proposed models, in which one can plug extraction processes from software artifacts and consistency checking techniques related to some of the consistency properties. Then, we plan to apply the models on other complex variability build systems, such as Busybox [21, 27], JHipster [14], and MozBuild [26], used on Firefox and other Mozilla projects. We expect the presented model and these extensions to enable practitioners, in the Linux community and in others, to incorporate the right consistency checking elements in their build system.

REFERENCES

[7] Cor-Paul Bezem, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. 2017. An empirical study of unspecified dependencies in...
Capturing the diversity of analyses on the Linux kernel variability