Capturing the diversity of analyses on the Linux kernel variability

Johann Mortara, Philippe Collet

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 → 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 [3], 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 [1]. 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 `FOO` (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_COND` needs to be satisfied. The selection of `FOO`, 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 `lib/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 `FOO` feature defined in the Kconfig file `lib/Kconfig`. `FOO` 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 `#elif` or `#else` can only be selected if the `ifdef` 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*, *IEEEXplore* and *Scopus*) searching for papers related to the consistency, build or configuration of the Linux kernel, and clearly

[1] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html#menu-structure

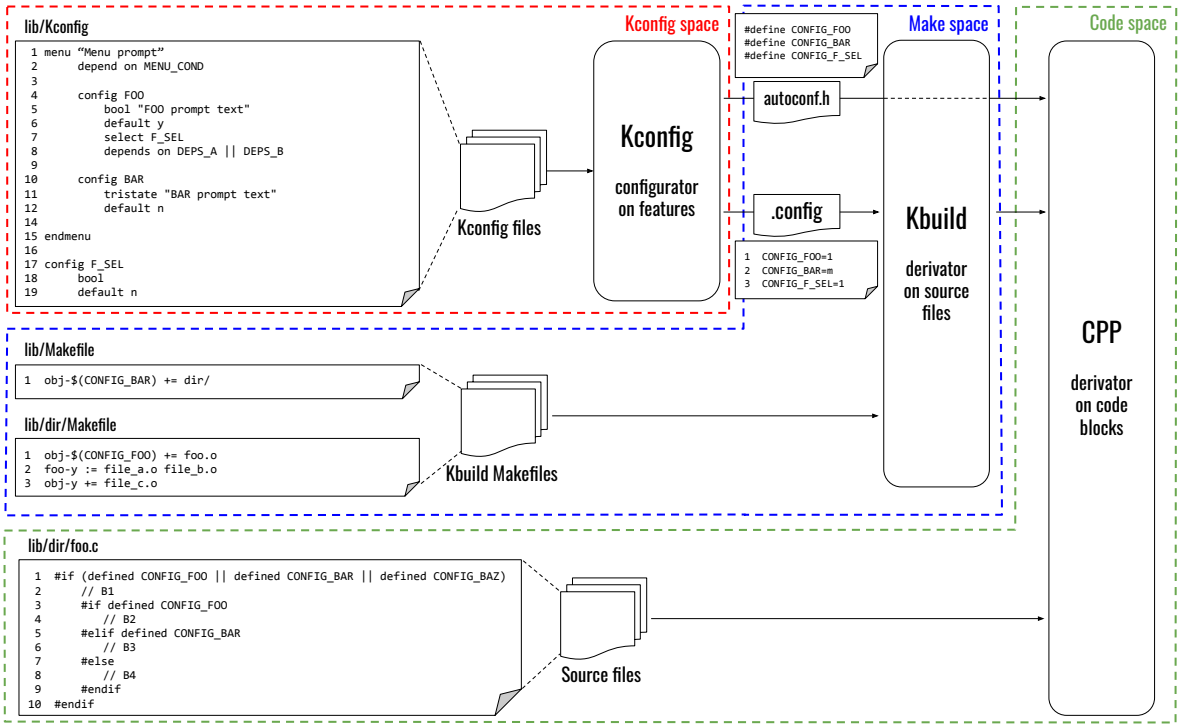

Figure 1: Sample Linux build process, inspired from [32] and [44]

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_i the

Presence Condition [44]:

$$\mathcal{PC}(b_i) = expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i)$$

with

$expression(b_i)$ Given a block b_i, the function $expression(b_i)$ returns the logical expression as specified in the block declaration. Example: For the block `B1` in fig. 1, the function $expression(b_1)$ returns: $A \vee B \vee C$.

$parent(b_i)$ Given b_i, $parent(b_i)$ 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.

$noPredecessors(b_i)$ Given b_i, $noPredecessors(b_i)$ 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 $noPredecessors(b_4)$ returns $\neg(b_2 \vee b_3)$.

The authors then give a definition of dead defect:

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

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

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. $satisfiable()$ represents the boolean satisfiability problem [2].

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

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:

$$\text{dead: } \neg sat(C \wedge \mathcal{I} \wedge Block_N)$$
$$\text{undead: } \neg sat(C \wedge \mathcal{I} \wedge \neg Block_N \wedge parent(Block_N))$$

with C and $\mathcal{I}$ the formulas representing the *configuration* (*i.e.*, KConfig) and *implementation* (*i.e.*, 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* $sat\ ((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$. We can infer the expressions for dead and undead configuration-implementation defects.

$$\text{dead: } \neg sat\ ((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$$
$$\text{undead: } \neg sat\ (\neg\,(b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V})$$

with $\mathcal{V}$ 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: $sat(feature \rightarrow presenceCondition(feature))$. However, no formal definition of $presenceCondition$ 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:*

$$sat\ ((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V} \wedge \neg\,(m_1 \vee \cdots \vee m_n))$$

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:

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge K)$$
$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge K)$$

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:

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge M \wedge K)$$
$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge M \wedge K)$$

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 space and the KConfig space".* These anomalies are checked by checking these formulas.

$$Dead_{F_N} = \neg sat(File_N \wedge M \wedge K)$$
$$Undead_{F_N} = \neg sat(\neg File_N \wedge M \wedge K)$$

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 Makefiles 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 papers are listed in table 1 [3]. Within just this subset, we can notice that every paper has its own terminology. To prevent the addition of yet another terminology to the state-of-the-art, we decided to reuse the terminology proposed by Nadi and Holt [31], as it is the only study we kept that covers the three spaces.

By analysing the definitions of anomalies from the selected papers, we can pinpoint multiple elements bringing confusion. First, multiple definitions are redundant between papers, but their expression and their names differ. For example, *Dead block* defined by Sincero et al. [44] in anom. {1}, *Configurability defects* defined by Tartler et al. [45] in anom. {13} and *Code–KConfig anomalies* defined by Nadi and Holt [31] in anom. {19} express the same formula. Moreover, spaces can also be named differently, sometimes with the same letter representing two different spaces in two definitions (*C* represents the CPP constraints in [31, 44] and the KConfig constraints in [45]). A summary of these differences is given in table 1.

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

Table 1: Notation and terminologies mapping for the three spaces in selected papers

Paper	KConfig files	KBuild Makefiles	CPP / Source files
[31]	Kconfig space	Make space	Code space
	K	**M**	**C**
[15]	Feature Modeling Configuration	Build system	Generator Preprocessor
[30]	Configuration space	Compilation space	Implementation space
[44]	Problem space Model	/	Solution space Implementation
	K	/	C
[45]	Configuration space	/	Implementation space
	C	/	I

On the opposite, some anomalies with identical names may not express the same type of defect. This is the case for the formulas to detect *dead* blocks in anom. {13} and anom. {15}, which are equivalent, while the characterizations of *undead* blocks are inconsistent. Let us take the following example:

```
1  #if defined A
2  //block 1
3  # if defined A
4      //block 2
5  # endif
6  #endif
```

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.

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

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:

- $terms(\phi)$ a helper function which, given a propositional formula, returns the terms in it (*e.g.*, $terms((A \wedge B) \vee C) = \{A, B, C\}$).
- $expand(\phi)$ a helper function which, given a propositional formula ϕ, replaces every asset a in ϕ by its presence condition, noted $\mathcal{PC}_{Int}(a)$ and defined in def. 2 (*e.g.*, $expand(b_1 \wedge \neg b_2) = \mathcal{PC}_{Int}(b_1) \wedge \neg \mathcal{PC}_{Int}(b_2)$).
- $slice(C, T)$ 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].

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

- ϕ_{select} is a propositional formula for the asset's selection ;
- ϕ_{preds} is a propositional formula on other assets that are evaluated before a. We call these assets *predecessors* ;
- ϕ_{depInt} is a propositional formula on assets on which a is dependent ;
- ϕ_{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:

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

- $file = \langle true, true, true, true \rangle$
- $b_1 = \langle FOO \vee BAR \vee BAZ, true, true, file \rangle$
- $b_2 = \langle FOO, true, b_1, file \rangle$
- $b_3 = \langle BAR, \neg b_2, b_1, file \rangle$
- $b_4 = \langle true, \neg(b_2 \vee b_3), b_1, file \rangle$

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

$$\mathcal{PC}_{Int}(a) = \phi_{select_a} \wedge expand\left(\phi_{preds_a}\right) \wedge expand\left(\phi_{depInt_a}\right)$$

[4]The principle of slicing has already been applied to feature models [2, 23] and its goal is to extract a subset of formulas equivalent to the whole space by keeping only formulas relevant to terms from T.

Note. An asset is selected if and only if its presence condition is satisfied: $\mathcal{PC}_{Int}(a) \leftrightarrow a$

Application to CPP. Let us take again the previous example.

$$\begin{aligned}
\mathcal{PC}_{Int}(b_1) &= \phi_{select_{b_1}} \wedge expand\left(\phi_{preds_{b_1}}\right) \wedge expand\left(\phi_{depInt_{b_1}}\right) \\
&= (FOO \vee BAR \vee BAZ) \wedge expand(true) \wedge expand(true) \\
&= (FOO \vee BAR \vee BAZ) \\
\mathcal{PC}_{Int}(b_2) &= \phi_{select_{b_2}} \wedge expand\left(\phi_{preds_{b_2}}\right) \wedge expand\left(\phi_{depInt_{b_2}}\right) \\
&= (FOO) \wedge true \wedge expand\,(b_1) \\
&= (FOO) \wedge (FOO \vee BAR \vee BAZ) \\
\mathcal{PC}_{Int}(b_3) &= \phi_{select_{b_3}} \wedge expand\left(\phi_{preds_{b_3}}\right) \wedge expand\left(\phi_{depInt_{b_3}}\right) \\
&= (BAR) \wedge (\neg \mathcal{PC}_{Int}(b_2)) \wedge \mathcal{PC}_{Int}(b_1) \\
\mathcal{PC}_{Int}(b_4) &= \phi_{select_{b_4}} \wedge expand\left(\phi_{preds_{b_4}}\right) \wedge expand\left(\phi_{depInt_{b_4}}\right) \\
&= (\neg(\mathcal{PC}_{Int}(b_2) \vee \mathcal{PC}_{Int}(b_3))) \wedge \mathcal{PC}_{Int}(b_1)
\end{aligned}$$

Note. Extracted presence conditions can be complex and may contain redundant terms (*e.g.*, $\mathcal{PC}_{Int}(b_2)$ is equivalent to FOO). Approaches to simplify presence conditions have been proposed [51] and are out of the scope of this paper.

Definition 3 (External presence condition)**.** By evaluating $\mathcal{PC}_{Int}$, we check that the asset can be selected given the constraints of its space. However, other external constraints may prevent the selection the asset. We call *context* the set of these constraints. The external presence condition of an asset in a given context C is defined as

$$\mathcal{PC}_{Ext}(a) = \mathcal{PC}_{Int}(a) \wedge slice(C, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a}))$$

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

$$C_{\text{Kconfig}} = \{FOO \rightarrow BAR, BAZ \rightarrow (\neg F1), F1 \rightarrow (\neg FOO), F3 \rightarrow F4\}$$
$$C_{\text{Kbuild}} = \{file \leftrightarrow FOO\}$$

then $\mathcal{PC}_{Ext}(b_1)$

$$\begin{aligned}
&= \mathcal{PC}_{Int}(b_1) \wedge slice(C, terms(\mathcal{PC}_{Int}(b_1)) \cup terms(\phi_{depExt_{b_1}})) \\
&= \mathcal{PC}_{Int}(b_1) \wedge slice(C, \{FOO, BAR, BAZ\} \cup \{file\}) \\
&= \mathcal{PC}_{Int}(b_1) \wedge ((FOO \rightarrow BAR) \wedge (BAZ \rightarrow (\neg F1)) \\
&\wedge (F1 \rightarrow (\neg FOO)) \wedge (file \leftrightarrow FOO))
\end{aligned}$$

4.1.1 Internal consistency. To express defects, we define dead, core, and full-mandatory assets, relying on definitions of dead and false-optional 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 $deads(\mathcal{A})$.

$$a \in deads(\mathcal{A}) \Leftrightarrow \neg sat(\mathcal{PC}_{Int}(a))$$

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.*, $expand(\phi_{depInt_s}) \rightarrow \neg\phi_{select_s}$) as in this case $\mathcal{PC}_{Int}(a)$ is inconsistent.

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

$$a \in core(\mathcal{A}) \Leftrightarrow \neg sat(\neg(\mathcal{PC}_{Int}(a)))$$

4.1.2 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 $deadsExt(\mathcal{A})$.

$$a \in deadsExt(\mathcal{A}) \Leftrightarrow \neg sat(\mathcal{PC}_{Ext}(a))$$

Definition 7 (Externally core asset)**.** An asset a of $\mathcal{A}$ 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 $coreExt(\mathcal{A})$.

$$a \in coreExt(\mathcal{A}) \Leftrightarrow \neg sat(\neg(\mathcal{PC}_{Ext}(a)))$$

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 $mandExt(\mathcal{A})$.

$$\begin{aligned} a \in mandExt(\mathcal{A}) &\Leftrightarrow expand(\phi_{depInt_a}) \rightarrow \mathcal{PC}_{Ext}(a) \\ &\Leftrightarrow \neg sat(\neg\mathcal{PC}_{Ext}(a) \wedge expand(\phi_{depInt_a})) \end{aligned}$$

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 assets of $\mathcal{A}$ with missing features is noted $missing(\mathcal{A})$.

$$a \in missing(\mathcal{A}) \Leftrightarrow \exists m \in terms(\mathcal{PC}_{Ext}(a)) \mid (m \notin terms(C))$$

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.

$F = \langle \phi_{enable}, \phi_{deps}, \mathcal{F}_{select} \rangle$

- ϕ_{enable} is a propositional formula representing the ability to select the feature ;
- ϕ_{deps} is a propositional formula on features on which F is dependent ;
- $\mathcal{F}_{select}$ is a set of features automatically selecting F. If a feature from $\mathcal{F}_{select}$ is selected, F is also selected, regardless of the precedent conditions.

Application to Kconfig. A feature F is a configuration option defined in a Kconfig file, with:

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

Table 2: Truth table for ϕ_{enable} from a Kconfig feature

Presence of `prompt`	Presence of `default`	ϕ_{enable}
yes	activated	$true$
	not activated	$true$
no	activated	$true$
	not activated	$false$

In the Kconfig file presented in fig. 1, three features are defined: `FOO`, `BAR` and `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 assets:

- $FOO = \langle true, (DEPS\_A \vee DEPS\_B) \wedge MENU\_COND, \{\} \rangle$
- $BAR = \langle true, MENU\_COND, \{\} \rangle$
- $F\_SEL = \langle false, true, \{FOO\} \rangle$

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.

$$\mathcal{PC}(F) = \Big(\phi_{enable} \wedge expand(\phi_{deps})\Big) \vee directSelect(F)$$

with $directSelect(F) = \bigvee_{F_s \in \mathcal{F}_{select}} \mathcal{PC}(F_s)$.

Note. The selection of a feature implies that its presence condition is satisfied: $F \rightarrow \mathcal{PC}(F)$ There is no biimplication 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.

$$\begin{aligned} \mathcal{PC}(FOO) &= (\phi_{enable_{FOO}} \wedge expand(\phi_{deps_{FOO}})) \vee directSelect(FOO) \\ &= true \wedge ((\mathcal{PC}(DEPS\_A) \vee \mathcal{PC}(DEPS\_B)) \wedge \mathcal{PC}(MENU\_COND)) \\ &= (\mathcal{PC}(DEPS\_A) \vee \mathcal{PC}(DEPS\_B)) \wedge \mathcal{PC}(MENU\_COND) \\ \mathcal{PC}(BAR) &= (\phi_{enable_{BAR}} \wedge expand(\phi_{deps_{BAR}})) \vee directSelect(BAR) \\ &= true \wedge \mathcal{PC}(MENU\_COND) \\ &= \mathcal{PC}(MENU\_COND) \\ \mathcal{PC}(F\_SEL) &= (false \wedge true) \vee \mathcal{PC}(FOO) = \mathcal{PC}(FOO) \end{aligned}$$

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 [5].

4.2.1 Consistency.

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

$$F \in deadFeatures() \Leftrightarrow \neg sat(\mathcal{PC}(F))$$

[5] For example, although Kconfig's syntax allows adding conditions to `select` statements, no defect described in our model requires to express this behaviour.

Definition 12 (Core feature)**.** A feature F of $\mathcal{F}$ is a core feature if it is always selected. The set of core features is noted $coreFeatures()$.

$$F \in coreFeatures() \Leftrightarrow \neg sat(\neg \mathcal{PC}(F))$$

Note. If $F_S \in \mathcal{F}_{select_F}$ 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 $missingDeadFeatures()$.

$$F \in missingDeadFeatures() \Leftrightarrow (m \in terms(\mathcal{PC}(F)) \wedge (m \notin \mathcal{F})$$

5 INSTANTIATION ON THE LINUX KERNEL

We now instantiate our model on the kernel build system. The configurator is used to model the Kconfig, while the derivator concept is used to model source files selected by the Kbuild Makefiles, with a form of positive variability [50]: the core is represented by the `obj-y` entries, where the additional parts are added in composite objects and feature dependent entries. The same derivator concept also represents the selection of code blocks from the source files by CPP, implementing this time negative variability [50].

5.1 Model on CPP

For CPP we have to describe the presence conditions and cross-space expressions for its related anomalies.

5.1.1 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]:

$$\mathcal{PC}_{Sin}(b_i) = expr(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i)$$

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.

$$\begin{aligned}\mathcal{PC}_{Sin}(b) &= expr(b) \wedge noPredecessors(b) \wedge parent(b) \\ &= \phi_{select_b} \wedge \neg\left(pred_1 \vee pred_2 \vee \cdots \vee pred_n\right) \wedge \phi_{depInt_b} \\ &= \phi_{select_b} \wedge \phi_{preds_b} \wedge \phi_{depInt_b}\end{aligned}$$

ϕ_{preds_b} and ϕ_{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.

$$\mathcal{PC}_{Sin}(b) = \phi_{select_b} \wedge expand\left(\phi_{preds_b}\right) \wedge expand(\phi_{depInt_b})$$

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.

$B_N \wedge C$. 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.

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

The Kconfig space K. K represents the set of constraints in the 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 $C = C_{\text{Kconfig}} \cup C_{\text{Kbuild}}$. Thus, $C_{\text{Kconfig}} = K$ and $C_{\text{Kbuild}} = M$, and:

$$\begin{aligned}slice(C_{\text{Kconfig}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models K \\ slice(C_{\text{Kbuild}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models M\end{aligned}$$

We can then express the different formulas in our model.

Instantiation 1 (Expressing code–Make–Kconfig anomalies {22})**.**

$$\begin{aligned}&\neg sat(B_N \wedge C \wedge M \wedge K) \\ &\qquad \Leftrightarrow \neg sat\left(\mathcal{PC}_{Int}(B_N) \wedge C\right) \\ &\qquad \Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(B_N)\right)\ (def.\ 3) \\ &\neg sat(\neg B_N \wedge parent(B_N) \wedge C \wedge M \wedge K) \\ &\qquad \Leftrightarrow \neg sat(C \wedge \neg \mathcal{PC}_{Int}(B_N) \wedge expand(\phi_{depInt_{B_N}})) \\ &\qquad \Leftrightarrow \neg sat\left(\neg \mathcal{PC}_{Ext}(B_N) \wedge expand(\phi_{depInt_{B_N}})\right)\end{aligned}$$

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

Instantiation 2 (Expressing configurability defects {13})**.** Same as inst. 1 with $C = C_{\text{Kconfig}}$.

Instantiation 3 (Expressing code–Kconfig {19})**.** Same as inst. 2.

Instantiation 4 (Expressing configuration-implementation defects {15})**.** $\mathcal{V}$ corresponds to the minimum but sufficient set of constraints from the configuration space. Thus:

$$sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right) \Leftrightarrow sat\left(\mathcal{PC}_{Int}(b_i) \wedge C_{\text{Kconfig}}\right)$$

We can then express *dead* and *undead* configuration-implementation defects. Given $\mathcal{B}$ the set of blocks and $C = C_{\text{Kconfig}}$:

$$\begin{aligned}\neg sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right) &\Leftrightarrow \neg sat\left(\mathcal{PC}_{Int}(b_i) \wedge C\right) \\ &\Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(b_i)\right)\ (def.\ 3) \\ \neg sat\left(\neg\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V}\right) &\Leftrightarrow \neg sat\left(\neg \mathcal{PC}_{Int}(b_i) \wedge C\right) \\ &\Leftrightarrow \neg sat\left(\neg \mathcal{PC}_{Ext}(b_i)\right)\ (def.\ 3)\end{aligned}$$

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 $B_N \wedge C \wedge M \wedge K \models \mathcal{PC}_{Ext}(B_N)$. If a feature m from the formula is not defined in the KConfig files, it means that $m \notin terms(K) \cup terms(M)$, *i.e.*, $m \notin terms(C_{\text{KConfig}} \cup C_{\text{KBuild}})$. Therefore: $\exists m \in terms(\mathcal{PC}_{Ext}(B_N)) \mid (m \notin terms(C))$, thus B_N is dead by missing feature.

Instantiation 6 (Expressing code–KConfig missing defects {20}). Same as inst. 5 with $C = C_{\text{KConfig}}$.

Instantiation 7 (Expressing referential defects {17}). If the feature is missing in the configuration space, then the definition corresponds def. 9 with $C = C_{\text{KConfig}}$ 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 $s = \langle \phi_{select}, \phi_{preds}, \phi_{depInt}, \phi_{depExt} \rangle$ 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.

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

Expressing the assets from fig. 1.

- $dir = \langle BAR, true, true, true \rangle$
- $foo = \langle FOO, true, dir, true \rangle$
- $file\_a = \langle true, foo, dir, true \rangle$
- $file\_b = \langle true, foo, dir, true \rangle$
- $file\_c = \langle true, true, dir, true \rangle$

Expressing their presence conditions

$$\begin{aligned}
\mathcal{PC}_{Int}(dir) &= \phi_{select_{dir}} \wedge expand\left(\phi_{preds_{dir}}\right) \wedge expand\left(\phi_{depInt_{dir}}\right) \\
&= BAR \wedge true \wedge true = BAR \\
\mathcal{PC}_{Int}(foo) &= FOO \wedge true \wedge expand(\mathcal{PC}_{Int}(dir)) = FOO \wedge BAR \\
\mathcal{PC}_{Int}(file_a) &= true \wedge expand(\mathcal{PC}_{Int}(foo)) \wedge expand(\mathcal{PC}_{Int}(dir)) \\
&= (FOO \wedge BAR) \wedge BAR \\
\mathcal{PC}_{Int}(file_b) &= \mathcal{PC}_{Int}(file_a) \\
\mathcal{PC}_{Int}(file_c) &= true \wedge true \wedge expand(\mathcal{PC}_{Int}(dir)) = BAR
\end{aligned}$$

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 ϕ_{select_a}, however is not present in the features defined in the KConfig files, obtained with $terms(C_{\text{KConfig}})$.

$$(m \in terms(\phi_{select_a})) \wedge (m \notin terms(C_{\text{KConfig}}))$$

As $terms(\phi_{select_{a_i}}) \subseteq terms(\mathcal{PC}_{Ext}(a_i))$, 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 $\phi_{preds_a} = comp$. a is an unused variable if $\neg\phi_{select_{comp}}$, and $\neg\phi_{select_{comp}} \rightarrow \neg\mathcal{PC}(comp) \rightarrow \neg expands(\phi_{preds_a}) \rightarrow \neg\mathcal{PC}(a)$. 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 $C = C_{\text{KConfig}}$.

$$\begin{aligned}
\mathcal{PC}_{Int}(s) &= \phi_{select_s} \wedge expand\left(\phi_{preds_s}\right) \wedge expand\left(\phi_{depInt_s}\right) \\
&= \phi_{select_s} \wedge \mathcal{PC}_{Int}(comp) \wedge \mathcal{PC}_{Int}(dir)
\end{aligned}$$

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 ($\bigvee f_i = \phi_{select_s}$), the composite object if present ($\mathcal{PC}_{Int}(comp)$) and its parent directory ($\mathcal{PC}_{Int}(dir)$) in the corresponding Makefiles. Therefore, $\mathcal{PC}_{Int}(s) \models (F_N \wedge M)$.

$$\begin{aligned}
\mathcal{PC}_{Ext}(s) &= \mathcal{PC}_{Int}(s) \wedge slice(C, terms(\mathcal{PC}_{Int}(s)) \cup terms(\phi_{depExt_s})) \\
&\models (F_N \wedge M) \wedge K \ (cf.\ section\ 5.1.2)
\end{aligned}$$

We can then express anom. {24} in our model:

$$\neg sat(F_N \wedge M \wedge K) \Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(s)\right)$$
$$\neg sat(\neg F_N \wedge M \wedge K) \Leftrightarrow \neg sat\left(\neg\mathcal{PC}_{Ext}(s)\right)$$

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 $C = C_{\text{KConfig}}$, 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 $\neg sat(\phi_{deps_F})$, which itself implies $\neg sat(\mathcal{PC}(F))$, 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:

$$(m \in terms(\phi_{deps_F})) \wedge (m \notin \mathcal{F})$$

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

Instantiation 15 (Expressing configuration-only defects {16}). The function $presenceCondition(feature)$ 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 $\phi_{enable_f} \wedge expand(\phi_{deps_f}) = \mathcal{PC}(f)$. Thus

$$\neg sat(f \rightarrow presenceCondition(f)) \Leftrightarrow \neg sat(\mathcal{PC}(f))$$

Therefore, f is dead.

Table 3: Anomalies covered by the model (defects defined as dead and undead according to the authors)

Paper			Sincero et al. [44]	Tartler et al. [45]	Nadi and Holt [30]	Nadi and Holt [31]	Hengelein [15]
Derivator	Internal consistency	Dead	anom. {2}	anom. {14}	anom. {12}	anom. {18}	
		Core	anom. {2}	anom. {14}		anom. {18}	
	External consistency	Dead	anom. {1,3}	anom. {13,15}		anom. {19,21,22,24}	
		Core	anom. {3}	anom. {15}		anom. {24}	
		Full-mandatory		anom. {13}		anom. {19,21,22}	
	Missing feature			anom. {17}	anom. {11}	anom. {20,23,25}	
Configurator	Dead			anom. {16}			anom. {4}
	Core						anom. {5}
	Missing dead						anom. {6}
Other properties (*e.g.*, unreachable symbol, file not used)					anom. {10}		anom. {7,8,9}

5.4 Resulting coverage

From the instantiations of the configurator model on Kconfig and the derivator on both Kbuild and 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 *full-mandatory* defect and the other as a *core* defect. Moreover, similar inconsistencies are exhibited in defects from [31], as well as two anomalies that are described as *dead* defects but are not called as such (anom. {12,16}).

6 THREATS TO VALIDITY

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.

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 Kconfig and 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 Kconfig, Kbuild, and 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 Kconfig, while the instantiated derivators can represent the Kbuild, selecting files, or 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

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 421–432.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2011. Slicing feature models. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 424–427.

[3] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*. IEEE, 114–123.

[4] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. 2002. Analyzing cloning evolution in the linux kernel. *Information and Software Technology* 44, 13 (2002), 755–765.

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information systems* 35, 6 (2010), 615–636.

[6] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines.. In *SPLC*. Citeseer, 498–499.

[7] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. 2017. An empirical study of unspecified dependencies in

make-based build systems. *Empirical Software Engineering* 22, 6 (2017), 3117–3148.

[8] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 21–30.

[9] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux kernel feature model changes using FMDiff. *Software & Systems Modeling* 16, 1 (2017), 55–76.

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

[11] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig translation to logic with one-way validation system. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 303–308.

[12] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM.

[13] Paul Gazzillo. 2017. Kmax: Finding all configurations of kbuild makefiles statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 279–290.

[14] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. 2017. Yo variability! JHipster: a playground for web-apps analyses. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. 44–51.

[15] Stefan Hengelein. 2015. *Analyzing the Internal Consistency of the Linux KConfig Model.* Master's thesis. University of Erlangen, Dept. of Computer Science.

[16] Ayelet Israeli and Dror G Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501.

[17] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast? case study on the linux kernel. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 101–110.

[18] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[19] Christian Kästner. 2016. KconfigReader. https://github.com/ckaestne/kconfigreader. Last access 24.02.2021.

[20] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. 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*. 805–824.

[21] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 773–792.

[22] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. Typechef: Toward type checking# ifdef variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. 25–32.

[23] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*. 60–64.

[24] Michael Larabel. 2020. The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019.

[25] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux kernel variability model. In *International Conference on Software Product Lines*. Springer, 136–150.

[26] Guillaume Maudoux and Kim Mens. 2019. Lessons and Pitfalls in Building Firefox with Tup. In *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019), Bolzano, Italy, July 8-10 Day, 2019 (CEUR Workshop Proceedings, Vol. 2510)*, Anne Etien (Ed.). CEUR-WS.org. http://ceur-ws.org/Vol-2510/sattose2019_paper_2.pdf

[27] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 50–61.

[28] Johann Mortara and Philippe Collet. 2021. *Capturing the diversity of analyses over the Linux kernel variability – Companion Technical Report.* https://doi.org/10.5281/zenodo.4715969

[29] Sarah Nadi. 2014. *Variability Anomalies in Software Product Lines.* Ph.D. Dissertation. University of Waterloo.

[30] Sarah Nadi and Ric Holt. 2011. Make it or break it: Mining anomalies from Linux Kbuild. In *2011 18th Working Conference on Reverse Engineering*. IEEE, 315–324.

[31] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 107–116.

[32] Sarah Nadi and Ric Holt. 2014. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process* 26, 8 (2014), 730–746.

[33] ThanhVu Nguyen and KimHao Nguyen. 2020. Using Symbolic Execution to Analyze Linux KBuild Makefiles. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 712–716.

[34] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. *ACM SigOps Operating Systems Review* 42, 4 (2008), 247–260.

[35] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. 2013. Coevolution of variability models and related artifacts: A case study from the Linux kernel. In *Proceedings of the 17th International Software Product Line Conference*. 91–100.

[36] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering* (2018).

[37] Andreas Ruprecht. 2015. *Lightweight Extraction of Variability Information from Linux Makefiles.* Master's thesis. Citeseer.

[38] Alcemir Rodrigues Santos and Eduardo Santana de Almeida. 2015. Do# ifdef-based Variation Points Realize Feature Model Constraints? *ACM SIGSOFT Software Engineering Notes* 40, 6 (2015), 1–5.

[39] Steven She. 2013. Linux Variability Analysis Tools. https://github.com/matachi/linux-variability-analysis-tools.exconfig. Last access 24.02.2021.

[40] Steven She and Thorsten Berger. 2010. Formal semantics of the Kconfig language. *Technical note, University of Waterloo* 24 (2010).

[41] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. *VaMoS* 10, 10 (2010), 45–51.

[42] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is the Linux Kernel a Software Product Line?. In *Proc. SPLC Workshop on Open Source Software and Product Lines*.

[43] Julio Sincero and Wolfgang Schröder-Preikschat. 2008. The Linux Kernel Configurator as a Feature Modeling Tool.. In *SPLC (2)*. Citeseer, 257–260.

[44] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 33–42.

[45] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*. 47–60.

[46] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[47] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*. 81–86.

[48] Thomas Thüm. 2020. A BDD for Linux? the knowledge compilation challenge for variability. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. 1–6.

[49] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. 2008. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software* 81, 6 (2008), 883–896.

[50] Markus Voelter and Iris Groher. 2007. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC 2007)*. IEEE, 233–242.

[51] Alexander Von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition simplification in highly configurable systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 178–188.

[52] Martin Walch, Rouven Walter, and Wolfgang Küchlin. 2015. Formal analysis of the Linux kernel configuration with SAT solving.. In *Configuration Workshop*. 131–138.

[53] Christoph Zengler and Wolfgang Küchlin. 2010. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, Vol. 2010. 51–56.

[54] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A propositional logic-based method for verification of feature models. In *International Conference on Formal Engineering Methods*. Springer, 115–130.