Capturing the diversity of analyses on the Linux kernel variability – Companion Technical Report

Johann Mortara, Philippe Collet

Capturing the diversity of analyses on the Linux kernel variability

Companion Technical Report - April 2021

JOHANN MORTARA, Université Côte d'Azur, CNRS, I3S, France
PHILIPPE COLLET, Université Côte d'Azur, CNRS, I3S, France

This technical report comes as a companion to *"Capturing the diversity of analyses on the Linux kernel variability"* [7] published at SPLC '21. In this report, we present additional inconsistencies in terminologies found in the state-of-the-art. We also give the exhaustive list of anomalies present in the five studied papers and detail the application of our formalism on all of them. Finally, we present a synthetic map of the studied anomalies over the Linux build system and their instantiations using our formalism.

Contents

1 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

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

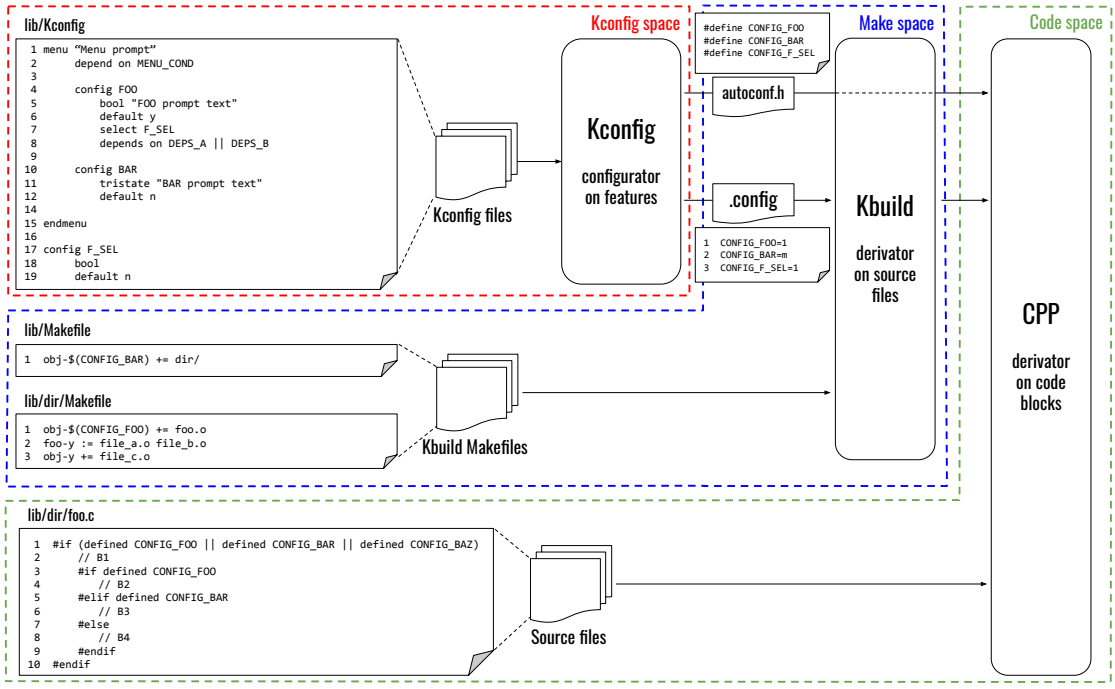

Fig. 1. Sample Linux build process, inspired from [11] and [14]

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

Table 1. Terminologies for each space of the Linux build system

Paper	KConfig files	KBuild Makefiles	CPP / Source files
Tartler et al. [18]	Model level	Generation level	Source code level
Tartler et al. [17]	Configuration space	Implementation variant	Implementation space
Nadi and Holt [10, 11]	Kconfig space	Make space	Code space
Hengelein [6], Tartler [15]	Feature Modeling Configuration	Build system	Generator Preprocessor
Passos et al. [12]	Variability Model	Mapping	Implementation
El-Sharkawy et al. [3]	Problem space	Solution space	
Abal et al. [1]	Problem space Model	/	Solution space Code
Nadi and Holt [9]	Configuration space	Compilation space	Implementation space
Nadi [8]	Configuration space	Build space	Code space
Sincero et al. [14]	Problem space Model	/	Solution space Implementation
Tartler et al. [16]	Configuration space	/	Implementation space

Chosen terminology

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).

2 DIVERSITY

Alongside the selected studies characterizing anomalies in the Linux build system, plethora of work describe it as they study some aspects of the Linux kernel (such as the evolution of its model [12]), or use it as a case study [1] or a benchmark for tools [3]. Table 1 summarizes, for the papers found by our query, the different terminologies used to refer to the different parts of the build system (namely, KConfig and its files, KBuild and its Makefiles, and CPP directives in source files). Except for a journal extension [8] and a PhD thesis [15], every paper has its own terminology, and some of them even use multiple terminologies ([1, 6, 15, 16]). One paper [3] groups the KBuild and CPP in a single *Solution space*, denomination used by Abal et al. [1] and Sincero et al. [14] to refer only to the CPP constraints.

In the papers we selected for study, we noticed redundant expressions of anomalies. However, the variables used to represent the different spaces diverge, hampering the comprehension of the expressed defects between the work. A summary of the different notations encountered is showed in table 2.

3 MODEL

As presented in the paper, deriving a variant of the Linux build system is done in three stages. First, a subset of the features defined in the KConfig files are selected, either by the user or by constraints, and form a valid configuration of

Table 2. Notation mapping for constraints in the three spaces

Paper	Properties	CPP	Make	Kconfig
Sincero et al. [14]	anom. {1}	C	/	K
Tartler et al. [16]	anom. {13}	I	/	C
Nadi and Holt [10]	anom. {19,21,22,24}	C	M	K

the Linux kernel. These features are then used in the Kbuild Makefiles to select the files from the code base which will be added for compilation, and by the C preprocessor (CPP) to select parts of these files.

3.1 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

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

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 [20] 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

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

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

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

3.1.1 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. [2], and full-mandatory features from Trinidad et al. [19].

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 $core(\mathcal{A})$.

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

3.1.2 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 $missing(\mathcal{A})$.

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

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 1, 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.

Table 3. 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	$false$

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

Expressing their presence conditions

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

3.2 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 3 ;
- 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 ;

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 [13] 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 [4]. Because of these limitations, the accuracy of variability reasoning approaches is also limited and acknowledged by researchers [5]. 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 [2].

3.2.1 *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 $deadFeatures()$.

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

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 $\mathcal{PC}(F_S) \rightarrow \mathcal{PC}(F)$.

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

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 $missingDeadFeatures()$.

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

3.2.2 Expressing cross-space formulas. Nadi and Holt [10] 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. [16]'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. [16] 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 [8], 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 3.1, 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:

$$slice(C_{\text{KConfig}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models K$$
$$slice(C_{\text{KBuild}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models M$$

4 DEFINITIONS

Presence Condition [14]:

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

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 $\neg(b_2 \vee b_3)$.

Compliance with presence conditions from Sincero et al. [14] For conciseness and to prevent confusion, we name this definition $\mathcal{PC}_{Sin}$ and use the more compact expression given in [16]:

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

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 3.1.

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

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.

Anomaly 1 (Dead block [14])**.** 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 [3].

Instantiation 1 (Dead block {1})**.** Same as inst. 13.

Anomaly 2 (Internal consistency [14])**.** Internal consistency is defined as *checking for each block of a compilation unit if it is selectable by at least one valid configuration.* This property is checked with $satisfiable(C_u \wedge b_i)$ which, expanded using the definition of C_u, gives:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i\right)$$

Instantiation 2 (Expressing Internal consistency {2})**.** $\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)$ corresponds to the set of constraints of the code space, and b_i the selection of the b_i block. Therefore, $\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i$ can be simplified to $b_i \leftrightarrow \mathcal{PC}(b_i)$, as done by Tartler et al. [16] in anom. {14}. Thus, anom. {2} $\Leftrightarrow$ anom. {14}.

Anomaly 3 (External consistency [14])**.** External consistency is defined as *checking for each block of a compilation unit if it is selectable by at least one valid configuration.* This property is checked with $satisfiable(C_u \wedge b_i \wedge FM)$ (with FM the representation of the feature model in a boolean formula) which, expanded using the definition of C_u, gives:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \wedge FM\right)$$

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

Instantiation 3 (Expressing External consistency {3}). In inst. 2, we showed:

$$\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \Leftrightarrow (b_i \leftrightarrow \mathcal{PC}(b_i))$$

FM in anom. {3} and $\mathcal{V}$ in anom. {15} both represent the KConfig space constraints. Therefore:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \wedge FM\right) \Leftrightarrow sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right)$$

i.e., anom. {3} $\Leftrightarrow$ anom. {15}.

Anomaly 4 (Dead feature [6]). A feature is dead if there are contradictions in its dependencies.

Instantiation 4 (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.

Anomaly 5 (False optional (undead) feature [6]). 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.

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

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

Instantiation 6 (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.

Anomaly 7 (Selects on Symbols with Dependencies [6]). `select` statements should not be used to select symbols matching the following conditions:

- The Symbol has dependencies
- The Symbol is selected by another symbol

Instantiation 7 (Expressing selects on symbols with dependencies {7}).

$$(terms(\phi_{deps_F}) \neq \emptyset) \wedge directSelect(F)$$

Anomaly 8 (Unreachable symbol [6]). A symbol is unreachable if:

- The symbol is invisible (does not have a prompt)
- The symbol is not selected by another symbol
- The symbol does not have a default value (or just default values with the value "n")

Instantiation 8 (Expressing unreachable symbol {8}). Given F a symbol. If the symbol does not have a prompt neither a default value allowing its selection, then $\neg sat(\phi_{enable_F})$. Selection by another feature is modeled with $directSelect(F)$. Thus:

$$\neg sat(\phi_{enable_F}) \wedge \neg directSelect(F)$$

Anomaly 9 (Unnecessary Selects on Choice Values [6])**.** `select` statements are unnecessary on symbols matching the following conditions:

- The Symbol is a choice value
- The Symbol is selected by another symbol

Instantiation 9 (Expressing unnecessary selects on choice values {9})**.** To express this defect, we need to add an extra *type* attribute to the feature. $type \in \{config, choice\}$ represents the way F is defined in the KCONFIG model, either as a simple $config$ element or in a *choice* statement.

$$(type_F = choice) \wedge directSelect(F)$$

Anomaly 10 (File Not Used (implementation-compilation consistency) [10])**.** A .c file exists in the directory but is not used in the Makefile of that directory.

Instantiation 10 (Expressing File Not Used {10})**.** This definition may not be valid anymore, since the syntax of KBUILD Makefiles allows them to explore subdirectories too [4]. However, we can generalise this definition with: *A .c file exists in the codebase but is not used in any Makefile.*

Given S the set of source files of the Linux kernel code base, and $\mathcal{A}_{\text{KBUILD}}$ the set of assets representing source files in the KBUILD Makefiles. A file $s \in S$ is a file not used if no asset in $\mathcal{A}_{\text{KBUILD}}$ corresponds to s:

$$\nexists a_i \in \mathcal{A}_{\text{KBUILD}} \mid s \equiv a_i$$

Anomaly 11 (Feature Not Defined (compilation-configuration consistency) [10])**.** 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.

Instantiation 11 (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 $terms(\phi_{selecta_i}) \subseteq terms(\mathcal{PC}_{Ext}(a_i))$, anom. {11} is a special case of def. 9, therefore a is a missing dead file.

Anomaly 12 (Variable Not Used (compilation self-consistency) [10])**.** A .c file is referenced in the Makefile as part of a composite variable definition, but this variable is never used.

Instantiation 12 (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.

Anomaly 13 (Configurability defect [16])**.** 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:

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

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

[4] https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#descending-down-in-directories

with C and $\mathcal{I}$ the formulas representing the *configuration* (*i.e.*, KCONFIG) and *implementation* (*i.e.*, Make) spaces respectively.

Instantiation 13 (Expressing configurability defects {13})**.** Same as inst. 22 with $C = C_{\text{KCONFIG}}$.

Anomaly 14 (Implementation-only defects [16], simplification of anom. {2})**.** *Implementation-only defects [...] represent code blocks that cannot be selected regardless of the systems' selected features; the structure of the source file itself contains contradictions that impede the selection of a block. This can be determined by checking the satisfiability of the formula* $sat\ (b_i \leftrightarrow \mathcal{PC}(b_i))$. We can infer the expressions for dead and undead implementation-only defects.

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

Instantiation 14 (Expressing Implementation-only defects {14})**.** Given $\mathcal{B}$ the set of blocks:

$$\begin{aligned} \neg sat\ (b_i \leftrightarrow \mathcal{PC}_{Sin}(b_i)) &\Leftrightarrow \neg sat\ (\mathcal{PC}_{Int}(b_i)) \\ &\Leftrightarrow b_i \in deads(\mathcal{B})\ (def.\ 4) \\ \neg sat\ (\neg\ (b_i \leftrightarrow \mathcal{PC}_{Sin}(b_i))) &\Leftrightarrow \neg sat\ (\neg\ (\mathcal{PC}_{Int}(b_i))) \\ &\Leftrightarrow b_i \in core(\mathcal{B})\ (def.\ 5) \end{aligned}$$

Anomaly 15 (Configuration-implementation defects [16], 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).

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

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

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

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

anom. {15} thus expresses dead (def. 6) and core defects (def. 7).

Anomaly 16 (Configuration-only defects [16])**.** 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.

Instantiation 16 (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.

Anomaly 17 (Referential defects [16])**.** *Referential defects are caused by a missing feature (m) that appears in either the configuration or the implementation space only. That is:*

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

is unsatisfiable.

Instantiation 17 (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 [16] does not consider this space, but it is not a defect for us.

Anomaly 18 (Code anomalies [10])**.** Code anomalies are defined as *"Conflicting code constraints"* and are not expressed in the paper as they are already determined by the UNDERTAKER tool designed in [16]. Thus, formulas to detect these anomalies are the ones from anom. {2} and anom. {14}.

Instantiation 18 (Expressing code anomalies {18})**.** Same as inst. 2.

Anomaly 19 (Code–KCONFIG [10])**.** 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.

Instantiation 19 (Expressing code–KCONFIG {19})**.** Same as inst. 13.

Anomaly 20 (Code–KCONFIG missing [10])**.** 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.*

Instantiation 20 (Expressing code–KCONFIG missing defects {20})**.** Same as inst. 23 with $C = C_{\text{KCONFIG}}$.

Anomaly 21 (Code–Make [10])**.** Code–Make anomalies are defined as *"Code constraints are not consistent with constraints in Makefiles"*. Although their formulas are not given in the paper, we can deduce them from anom. {19}:

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

Instantiation 21 (Expressing code–Make defects {21}). Same as inst. 22 with $C = C_{\text{KBUILD}}$.

Anomaly 22 (Code–Make–KCONFIG [10]). Code–Make–KCONFIG anomalies are defined as *"The combination of constraints in the three spaces are conflicting"* and detected using the following formulas:

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

Instantiation 22 (Expressing code–Make–KCONFIG anomalies {22}).

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

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

Anomaly 23 (Code–Make–KCONFIG missing [10]). 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"*.

Instantiation 23 (Expressing code–Make–KCONFIG missing defects {23}). We showed in inst. 22 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.

Anomaly 24 (Make–KCONFIG [10]). 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.

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

Instantiation 24 (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 [10] 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

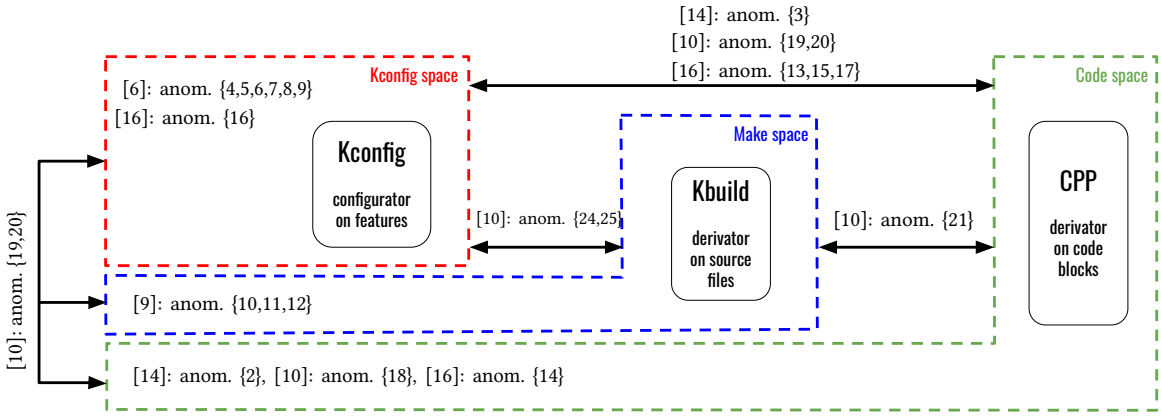

Fig. 2. Synthetic map of inconsistencies analyses of the Linux Kernel

($\mathcal{PC}_{Int}(comp)$) and its parent directory ($\mathcal{PC}_{Int}(dir)$) in the corresponding Makefiles. Therefore, $\mathcal{PC}_{Int}(s) \models (F_N \wedge M)$.

$$\mathcal{PC}_{Ext}(s) = \mathcal{PC}_{Int}(s) \wedge slice(C, terms(\mathcal{PC}_{Int}(s)) \cup terms(\phi_{depExt_s}))$$
$$\models (F_N \wedge M) \wedge K \ (cf.\ section\ 3.2.2)$$

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

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

anom. {24} thus expresses dead (def. 6) and core defects (def. 7).

Anomaly 25 (Make–Kconfig missing [10])**.** 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.

Instantiation 25 (Expressing Make–Kconfig missing defects {25})**.** Same as inst. 23 with $C = C_{\text{Kconfig}}$, and relying on formulas from inst. 24.

5 RESULTING TABLES

We summarize our study of the existing work on anomalies in the Linux build system by providing a synthetic map illustrates the coverage of the different papers and the anomalies they express over the different spaces of the Linux build system fig. 2. Table 4 describes for every anomaly the expressed property in our formalism.

Table 4. The studied anomalies over the Linux build system and their instantiation using our model

Paper	Defect	Derivator Internal consistency		
		Dead	**Core**	
[14]	Dead block (anom. {1} and inst. 1)	✓		
	Internal consistency (anom. {2} and inst. 2)	✓	✓	
[10]	Code anomalies (anom. {18} and inst. 18)	✓	✓	
[16]	Implementation-only defects (anom. {14} and inst. 14)	✓	✓	
[9]	Variable Not Used (anom. {12} and inst. 12)	✓		
		Derivator External consistency		
		Dead	**Core**	**Full-mandatory**
[14]	External consistency (anom. {3} and inst. 3)	✓	✓	
[10]	Code–Kconfig anomalies (anom. {19} and inst. 19)	✓		✓
	Code–Make anomalies (anom. {21} and inst. 21)	✓		✓
	Code–Make–Kconfig anomalies (anom. {22} and inst. 22)	✓		✓
	Make–Kconfig anomalies (anom. {24} and inst. 24)	✓	✓	
[16]	Configurability defects (anom. {13} and inst. 13)	✓		✓
	Configuration-implementation defects (anom. {15} and inst. 15)	✓	✓	
		Derivator missing feature		
[10]	Code–Kconfig missing anomalies (anom. {20} and inst. 20)	✓		
	Code–Make–Kconfig missing anomalies (anom. {23} and inst. 23)	✓		
	Make–Kconfig missing anomalies (anom. {25} and inst. 25)	✓		
[9]	Feature Not Defined (anom. {11} and inst. 11)	✓		
[16]	Referential defects (anom. {17} and inst. 17)	✓		
		Configurator consistency		
		Dead	**Missing dead**	**Core**
[16]	Configuration-only defects (anom. {16} and inst. 16)	✓		
[6]	Dead feature (anom. {4} and inst. 4)	✓		
	Missing dead feature (anom. {6} and inst. 6)		✓	
	False optional (undead) feature (anom. {5} and inst. 5)			✓
		Other properties		
[9]	File Not Used (anom. {10} and inst. 10)	✓		
[6]	Unreachable Symbols (anom. {8} and inst. 8)	✓		
	Unnecessary Selects on Choice Values (anom. {9} and inst. 9)	✓		
	Selects on Symbols with Dependencies (anom. {7} and inst. 7)	✓		

Table 5. Anomalies covered by the model (defects defined as dead and undead according to the authors)

Paper			Sincero et al. [14]	Tartler et al. [16]	Nadi and Holt [9]	Nadi and Holt [10]	Hengelein [6]
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}

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] 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.

[3] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. 19–28.

[4] 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.

[5] 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.

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

[7] Johann Mortara and Philippe Collet. 2021. Capturing the diversity of analyses on the Linux kernel variability. In *25th ACM International Systems and Software Product Line Conference - Volume A (SPLC '21)*.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] Reinhard Tartler. 2013. *Mastering variability challenges in Linux and related highly-configurable system software.* Ph.D. Dissertation.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.