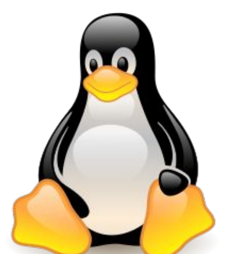
Capturing the diversity of analyses on the Linux kernel variability

Johann Mortara – Philippe Collet

Université Côte d'Azur, CNRS, I3S, France

SPLC '21
September 10, 2021

The Linux kernel



Highly configurable operating system

Some statistics:

- **15K+** features
- **28M+** LoC in **60K+** files
- **900K+** commits by **2K+** contributors

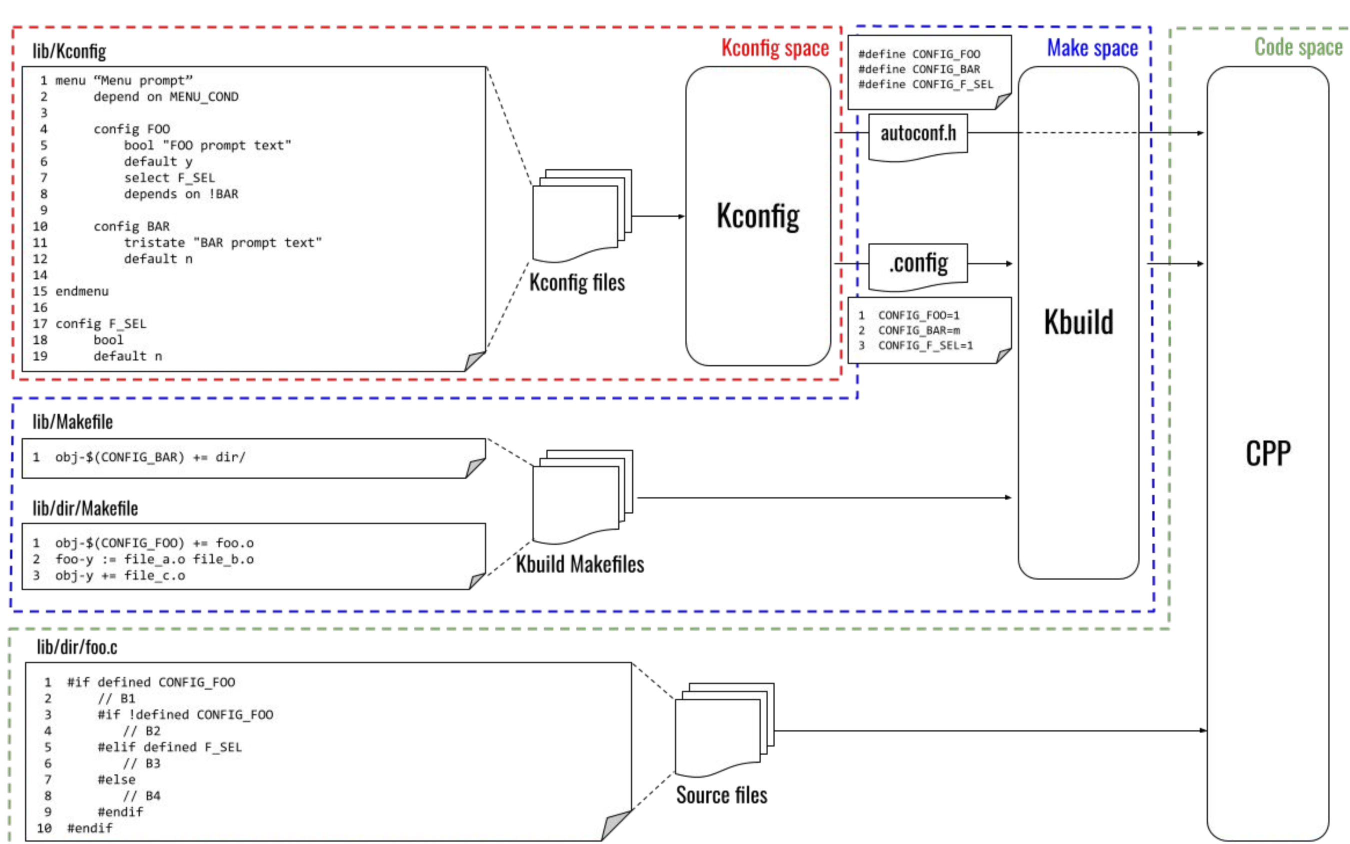
Used as a case study by plethora of research work in different domains:

- security
- code quality
- development process
- ...
- and also in the SPL field!

The Linux build system

3 steps:

1. Kconfig: selection of features
2. Kbuild: selection of source files
3. CPP: selection of code blocks



Linux build system anomalies

An **anomaly** defines a **property** describing a **defect** in the build system.

Each defect is formalized as a **satisfiability check** on a **boolean formula** to check for the presence of the anomaly.

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:

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

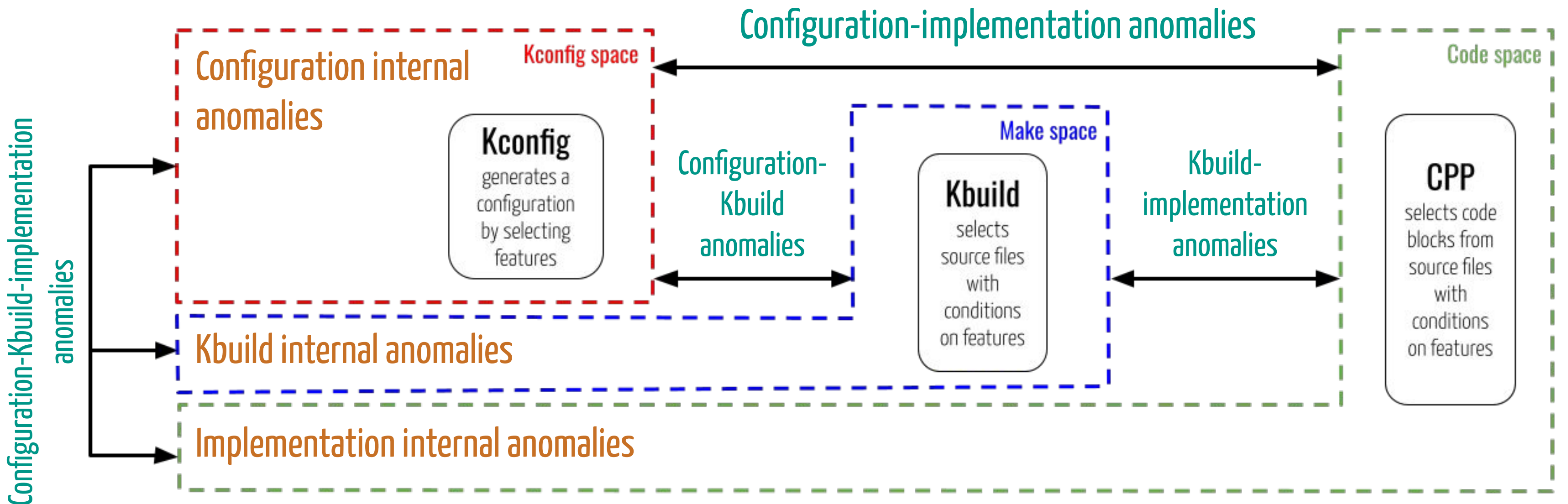
$$\text{undead: } \neg sat(C \wedge \mathcal{I} \wedge \neg Block_N \wedge parent(Block_N))$$

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

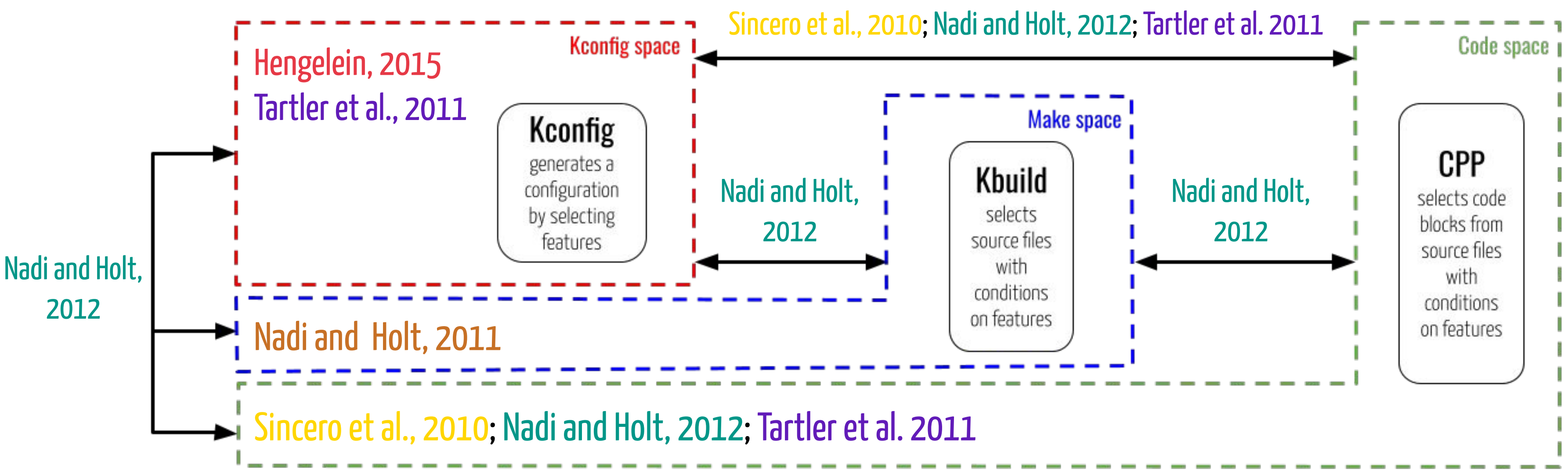
Studied anomalies in the Linux build system

Internal anomalies

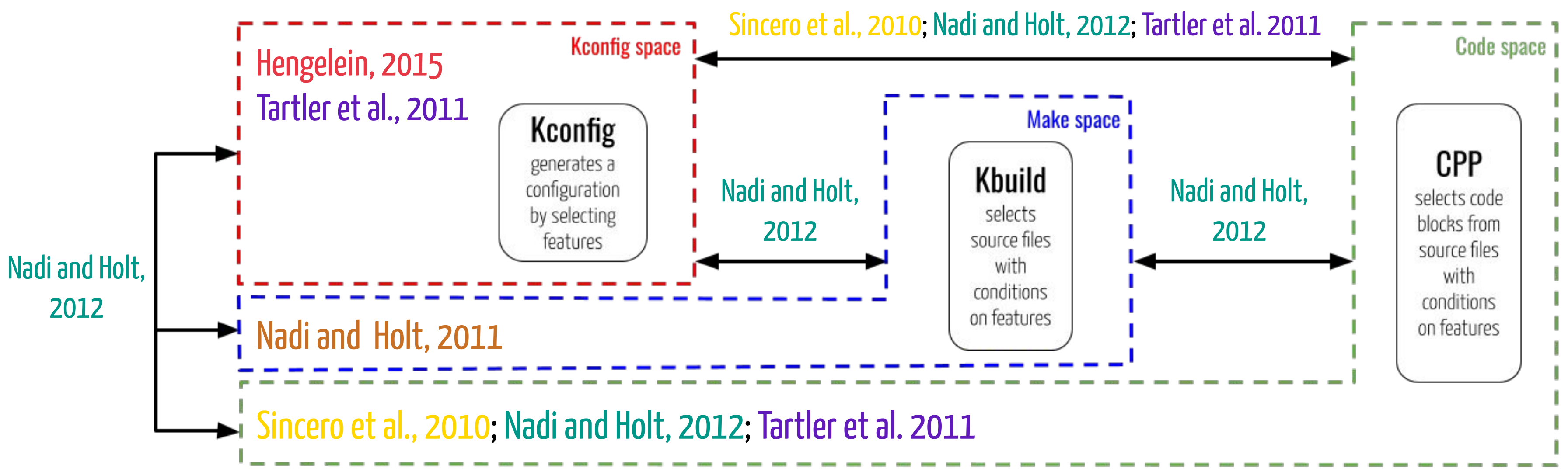
External anomalies



No existing formalism covers the whole set of anomalies in the build system

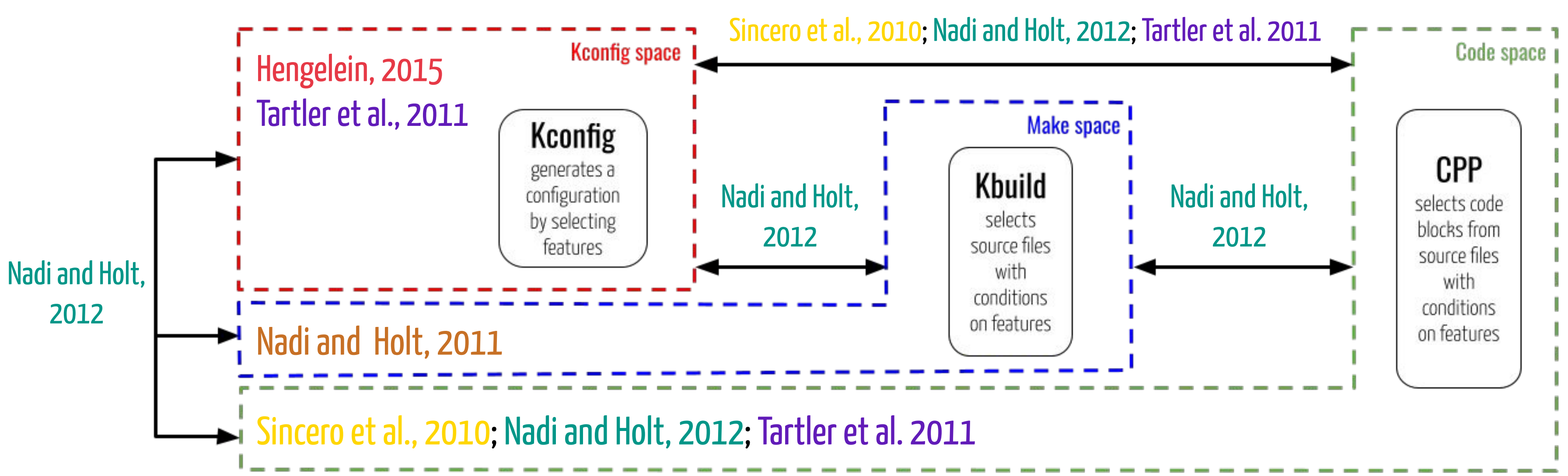


No existing formalism covers the whole set of anomalies in the build system



Partial view of the anomalies

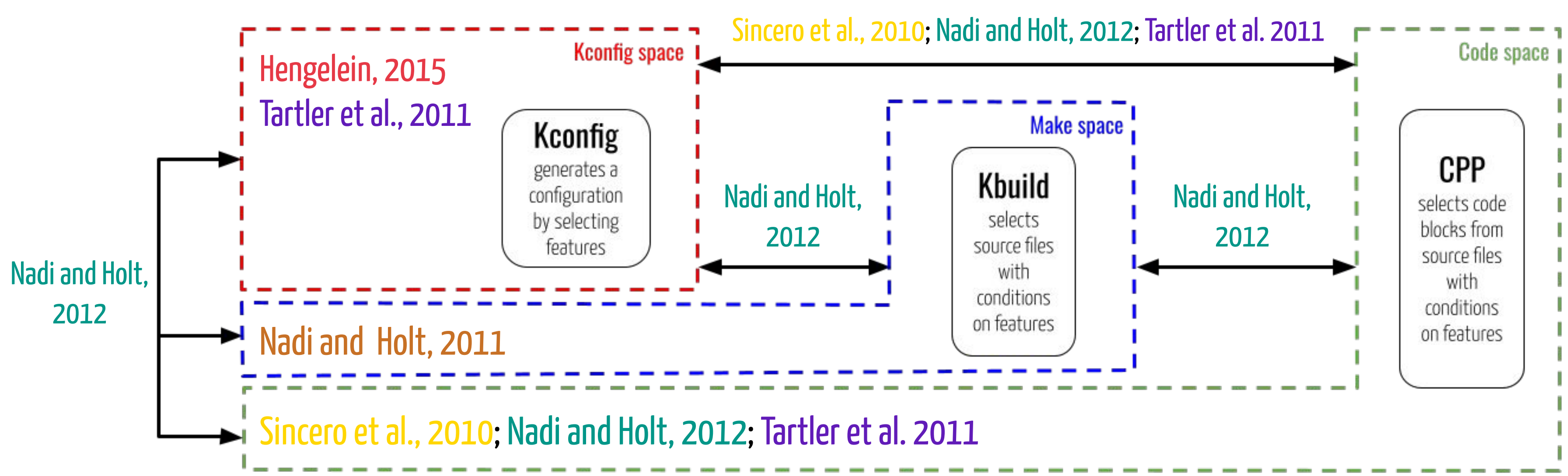
No existing formalism covers the whole set of anomalies in the build system



Partial view of the anomalies

Different denominations for the three spaces

No existing formalism covers the whole set of anomalies in the build system



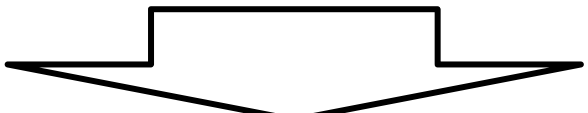
Partial view of the anomalies

Different denominations for the three spaces

Overlapping formulas with different conventions

Partial view ≠ denominations ≠ conventions

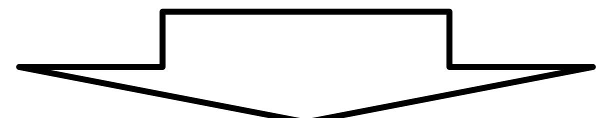
Partial view ≠ denominations ≠ conventions



Unified model Global view

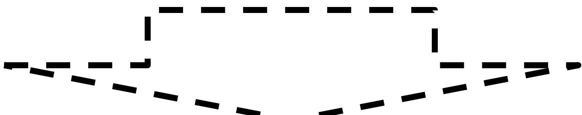
Fine-grained vision Single terminology

Partial view ≠ denominations ≠ conventions



Unified model Global view

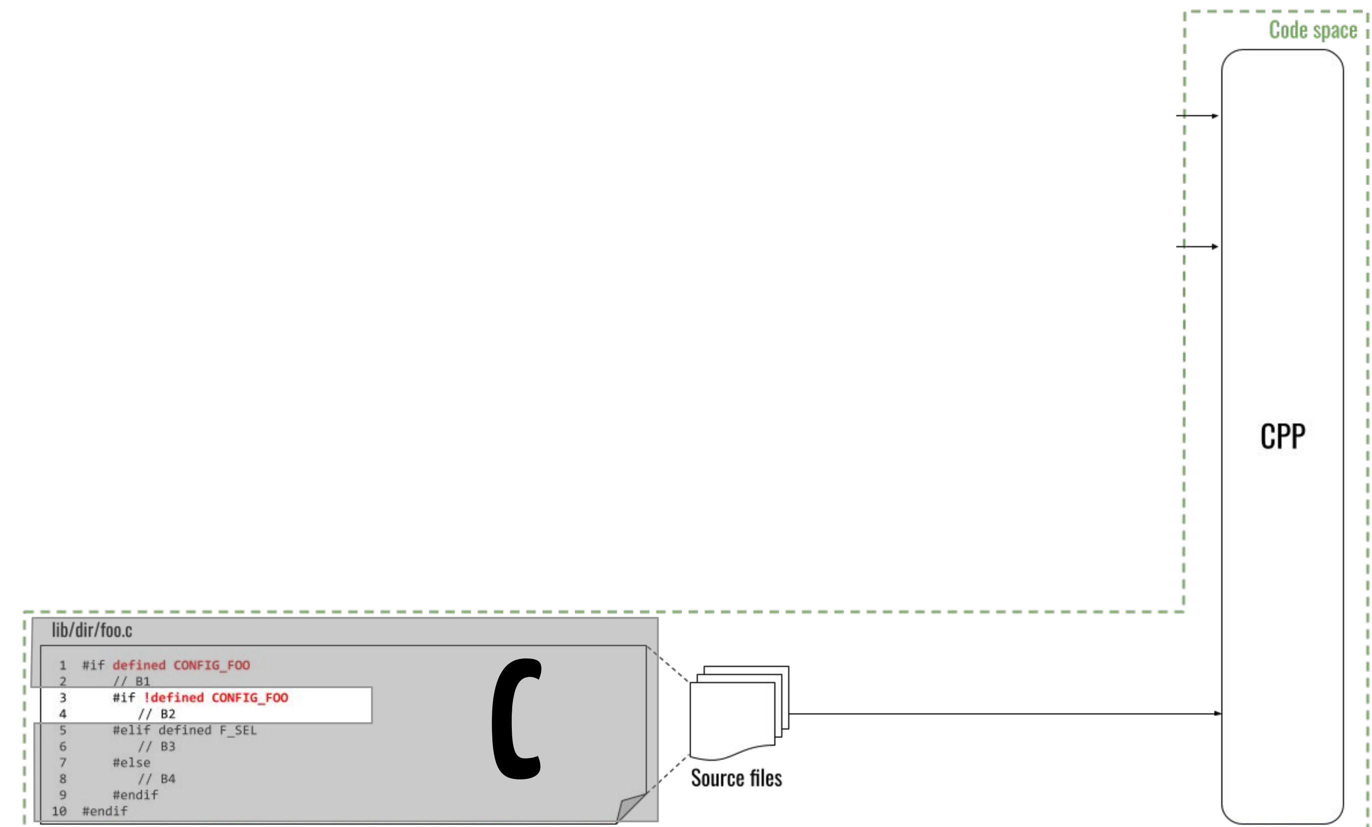
Fine-grained vision Single terminology



Better comprehension Applicability

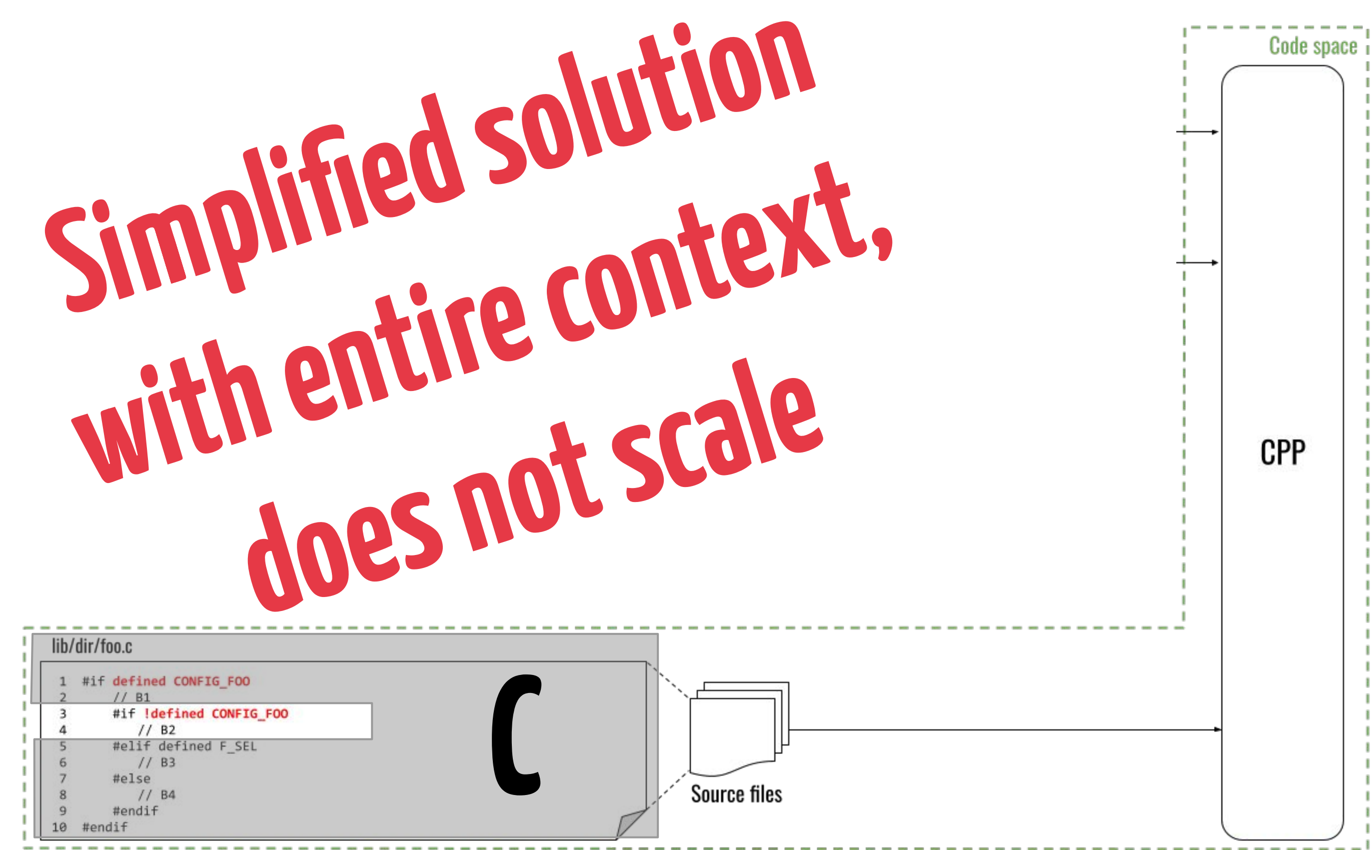
Example of internal anomaly (Code space) [Sincero et al., 2010]

B2 dead ⇔
¬sat(B2 ∧ C)

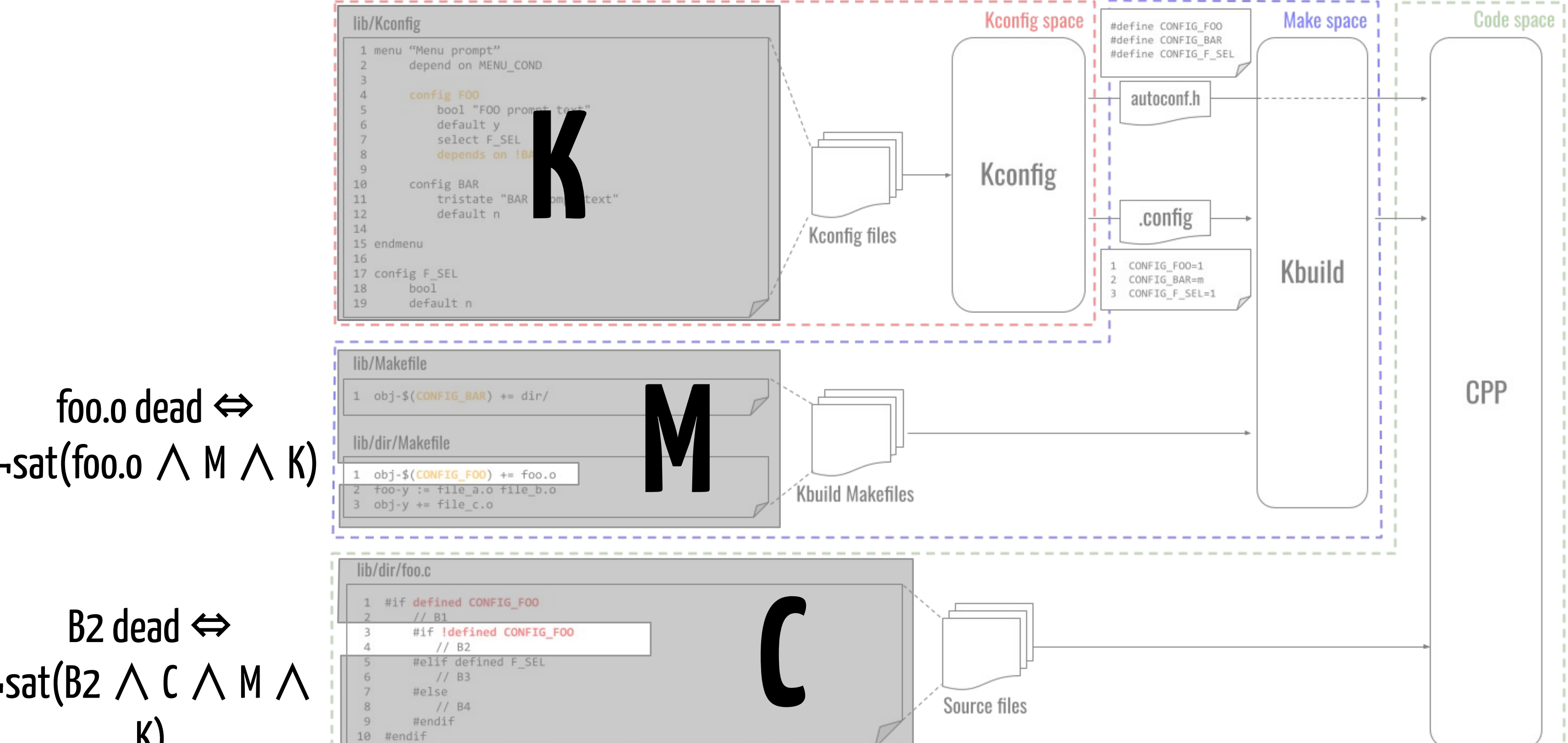


Example of internal anomaly (Code space) [Sincero et al., 2010]

B2 dead ⇔ ¬sat(B2 ∧ C)



Example of external anomaly (Code and Make spaces) [Nadi and Holt, 2012]



Example of external anomaly (Code and Make spaces) [Nadi and Holt, 2012]

Incoherences

Tartler et al., 2011:

"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****."*

⇒ block2 undead

Example:

```
1  #if defined A
2    // block1
3    #if defined A
4      // block2
5    #endif
6  #endif
```

Incoherences

Tartler et al., 2011:

“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****.”*

⇒ block2 undead

“Defects appear in two ways, either as dead, that is, unselectable blocks, or ***undead, that is, always present blocks****.”*

⇒ block2 not undead

Example:

```
1  #if defined A
2    // block1
3    #if defined A
4      // block2
5    #endif
6  #endif
```

Identical defect name but different semantics

Incoherences

Different denominations for the three spaces

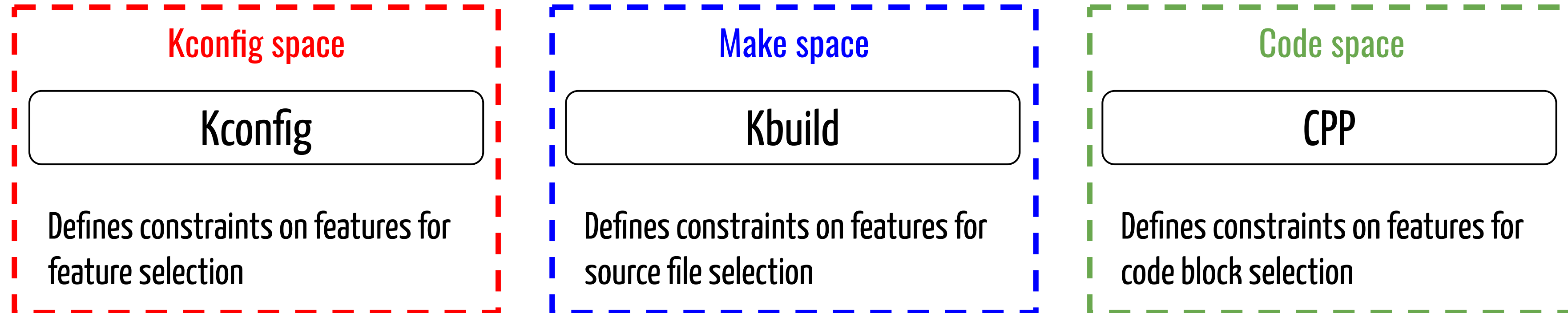
Paper	KCONFIG files	KBUILD Makefiles	CPP / Source files
Tartler et al. [17]	Model level	Generation level	Source code level
Tartler et al. [16]	Configuration space	Implementation variant	Implementation space
Nadi and Holt [9, 10]	Kconfig space	Make space	Code space
Hengelein [6], Tartler [14]	Feature Modeling Configuration	Build system	Generator Preprocessor
Passos et al. [11]	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 [8]	Configuration space	Compilation space	Implementation space
Nadi [7]	Configuration space	Build space	Code space
Sincero et al. [13]	Problem space Model	/	Solution space Implementation
Tartler et al. [15]	Configuration space	/	Implementation space

Chosen terminology

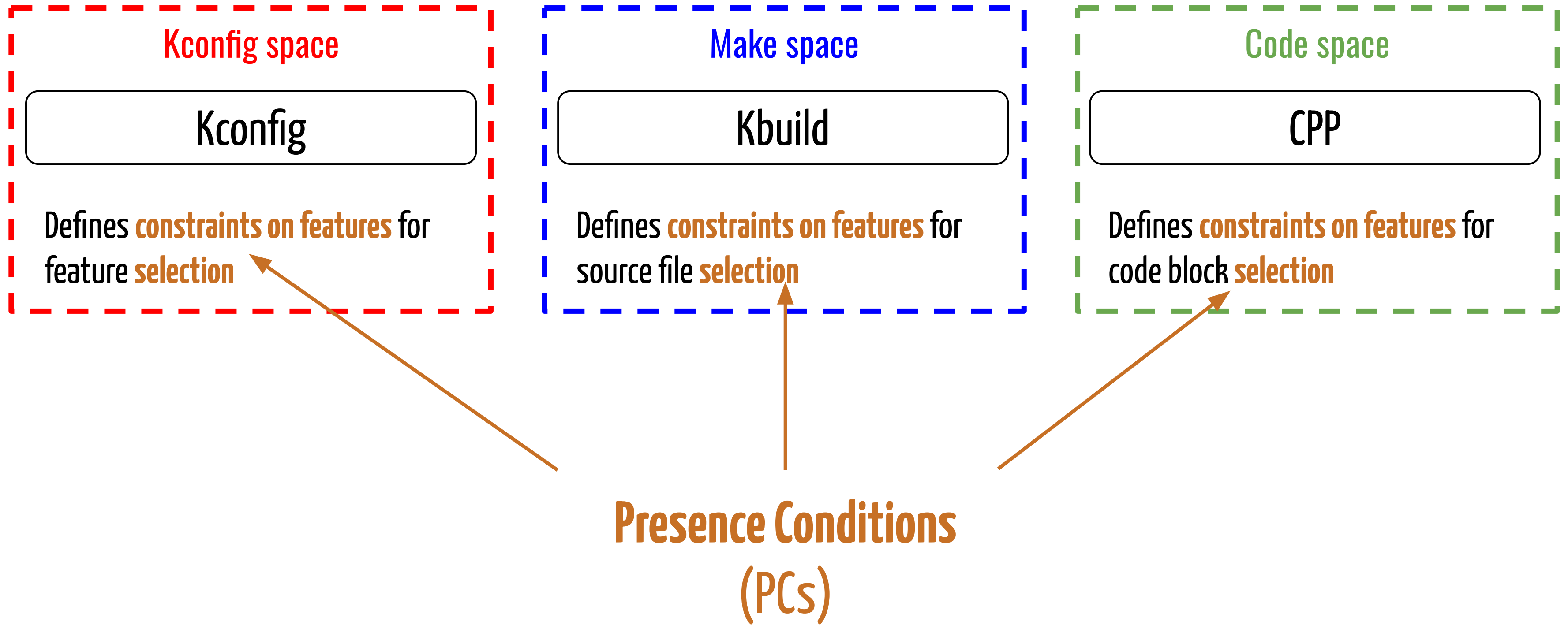
Overlapping formulas with different conventions

Paper	Properties	CPP	Make	KCONFIG
Sincero et al. [13]	anom. {1}	C	/	K
Tartler et al. [15]	anom. {13}	I	/	C
Nadi and Holt [9]	anom. {19,21,22,24}	C	M	K

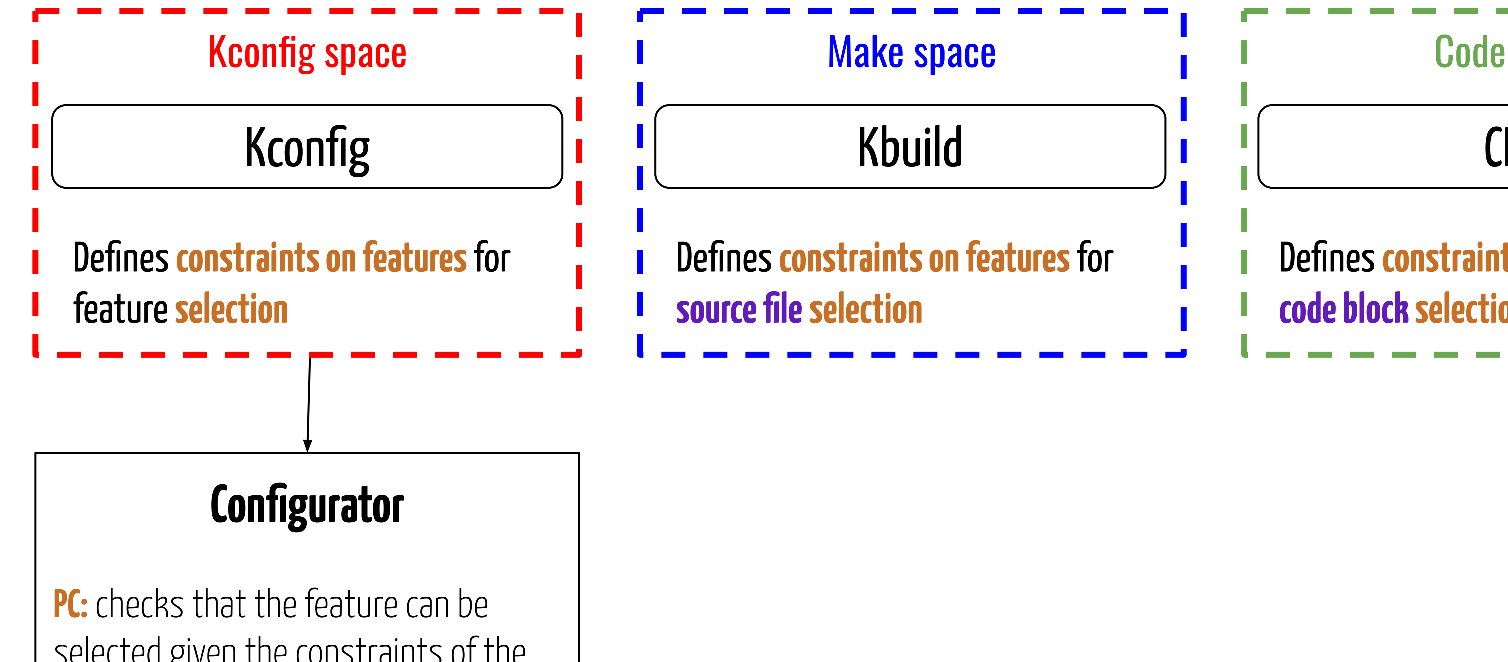
Contribution: design choices



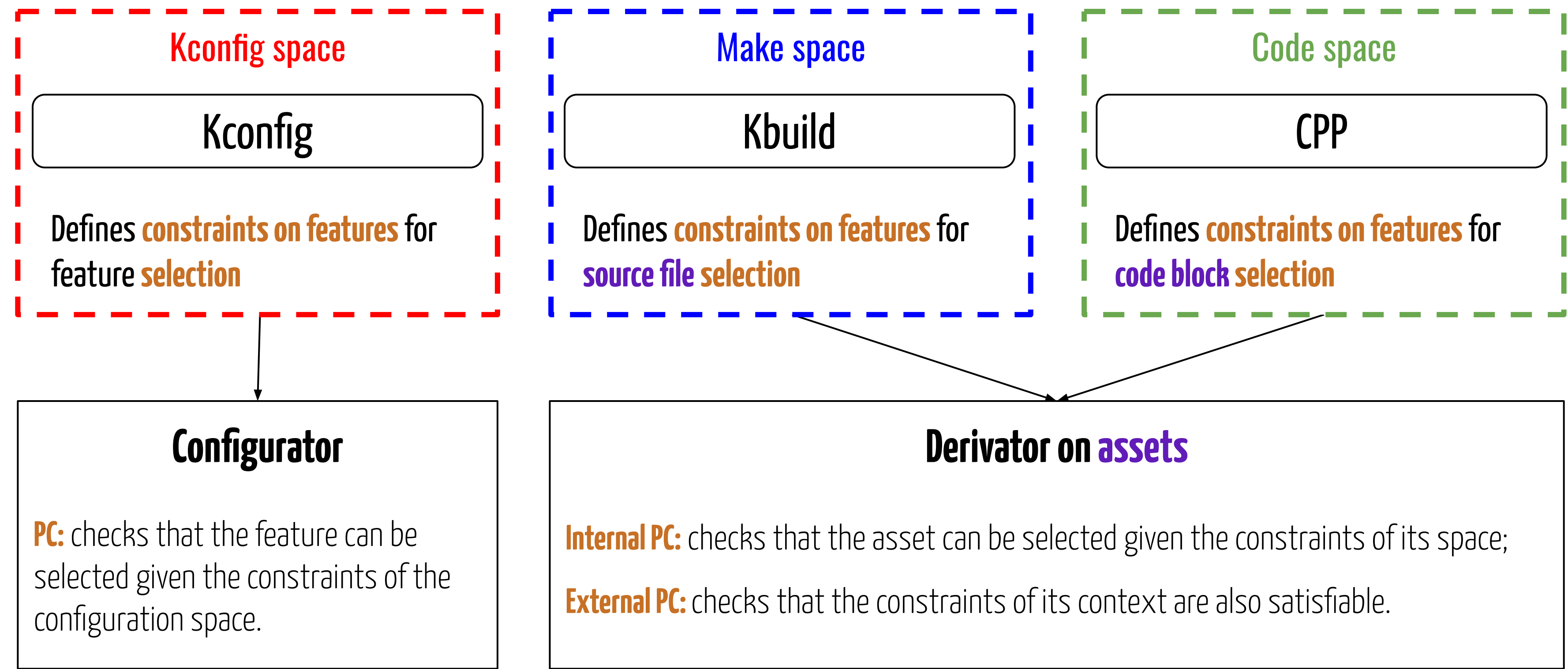
Contribution: design choices



Contribution: design choices



Contribution: design choices



Model concepts

Kconfig
Configurator

Make
Derivator

CPP
Derivator

Model concepts

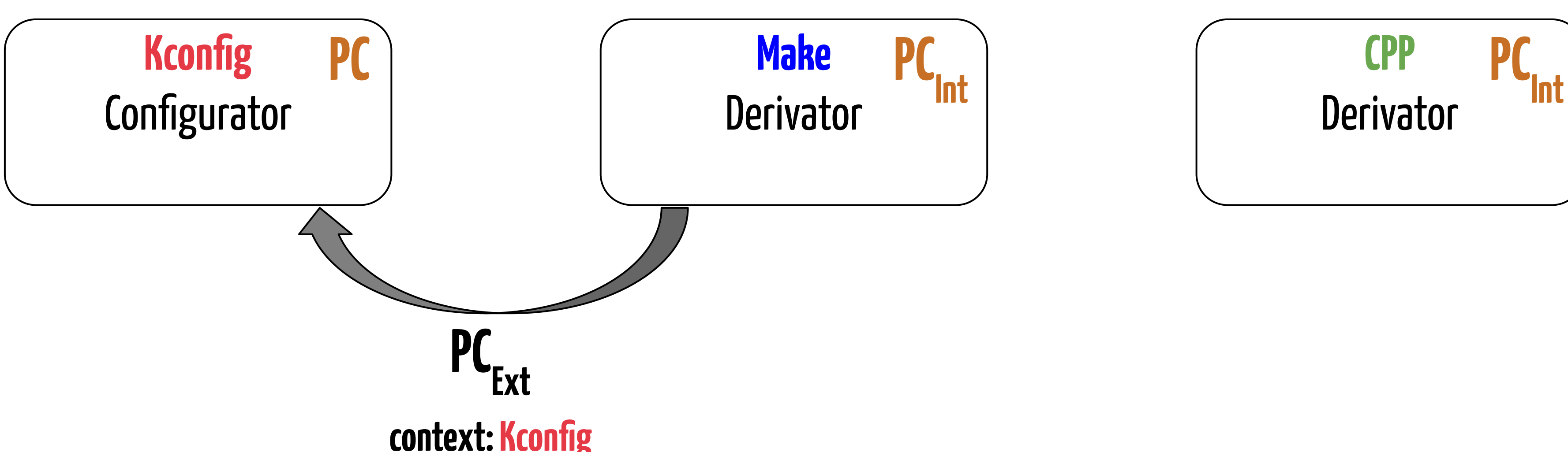
Kconfig PC
Configurator

Make PC_{Int}
Derivator

CPP PC_{Int}
Derivator

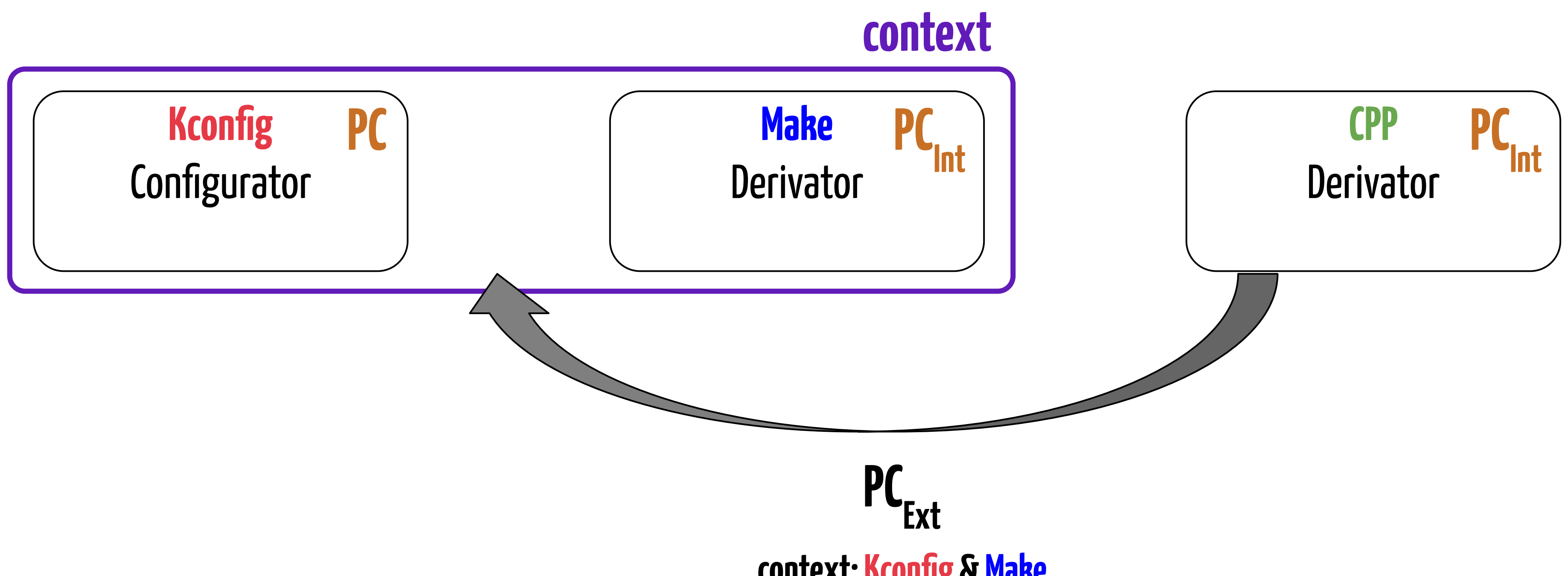
PC = Presence Condition

Model concepts

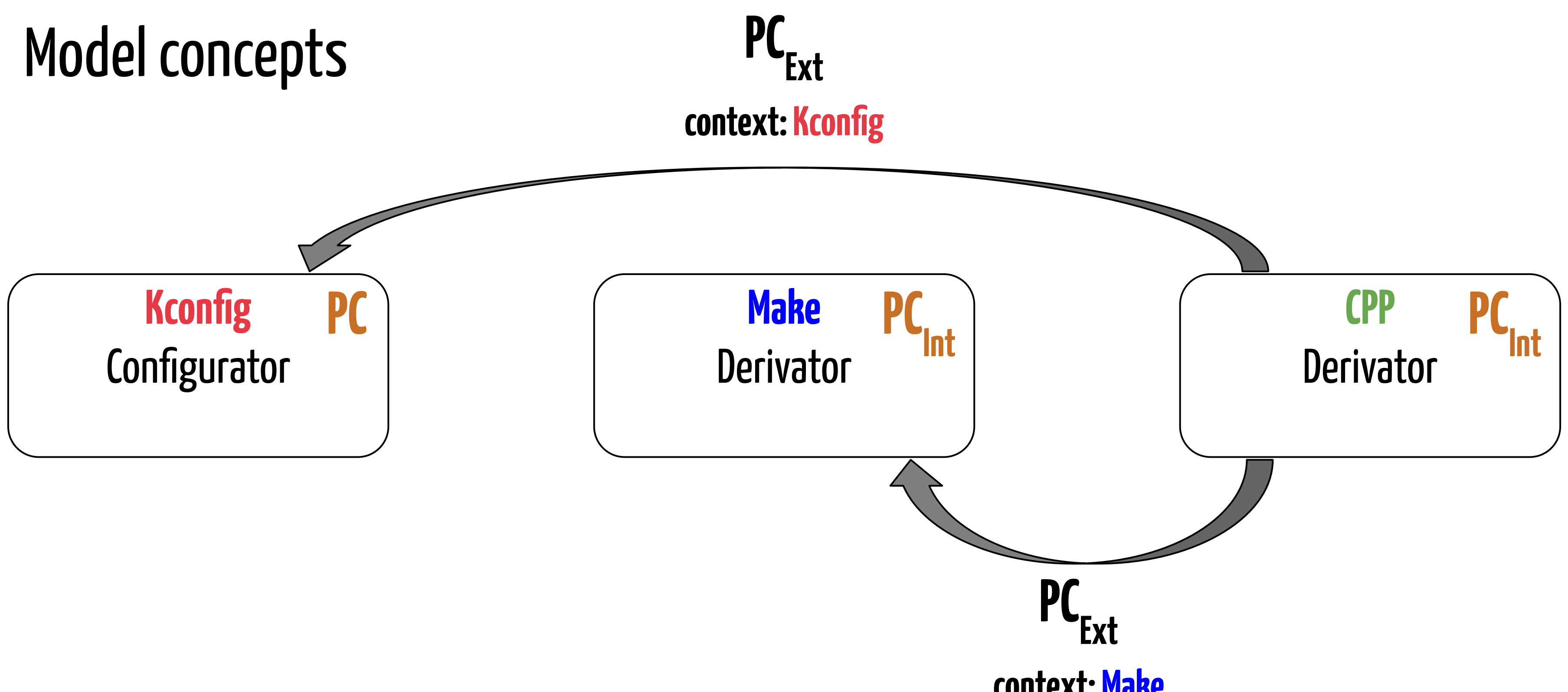


PC = Presence Condition

Model concepts

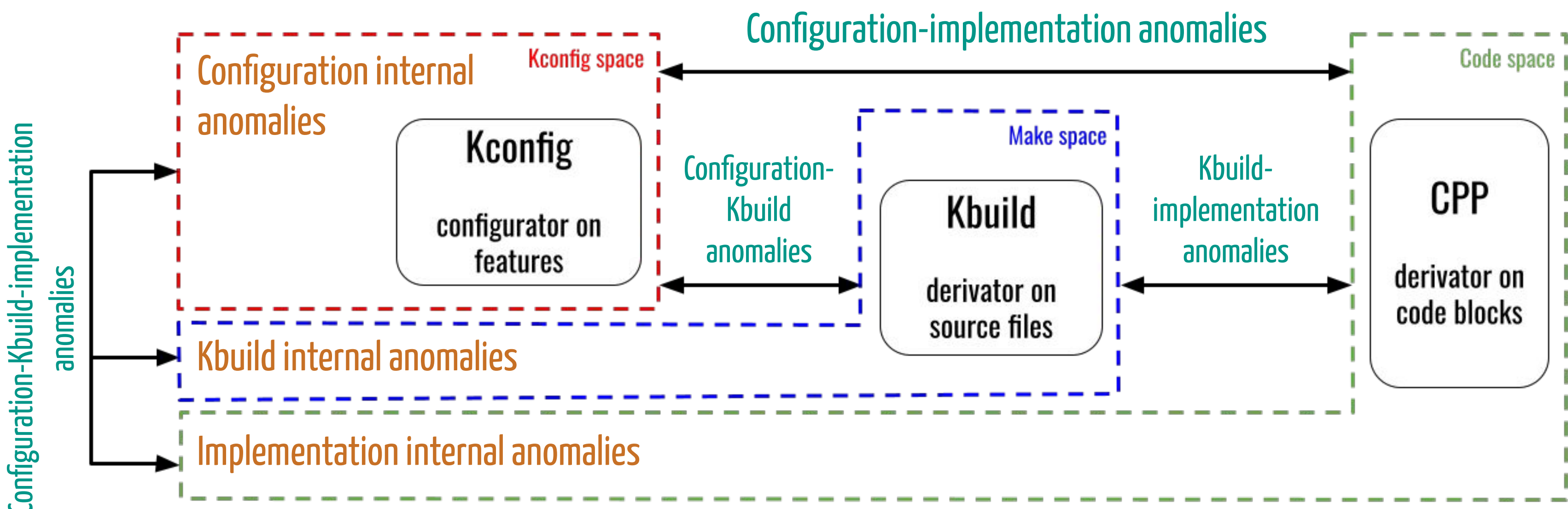


PC = Presence Condition



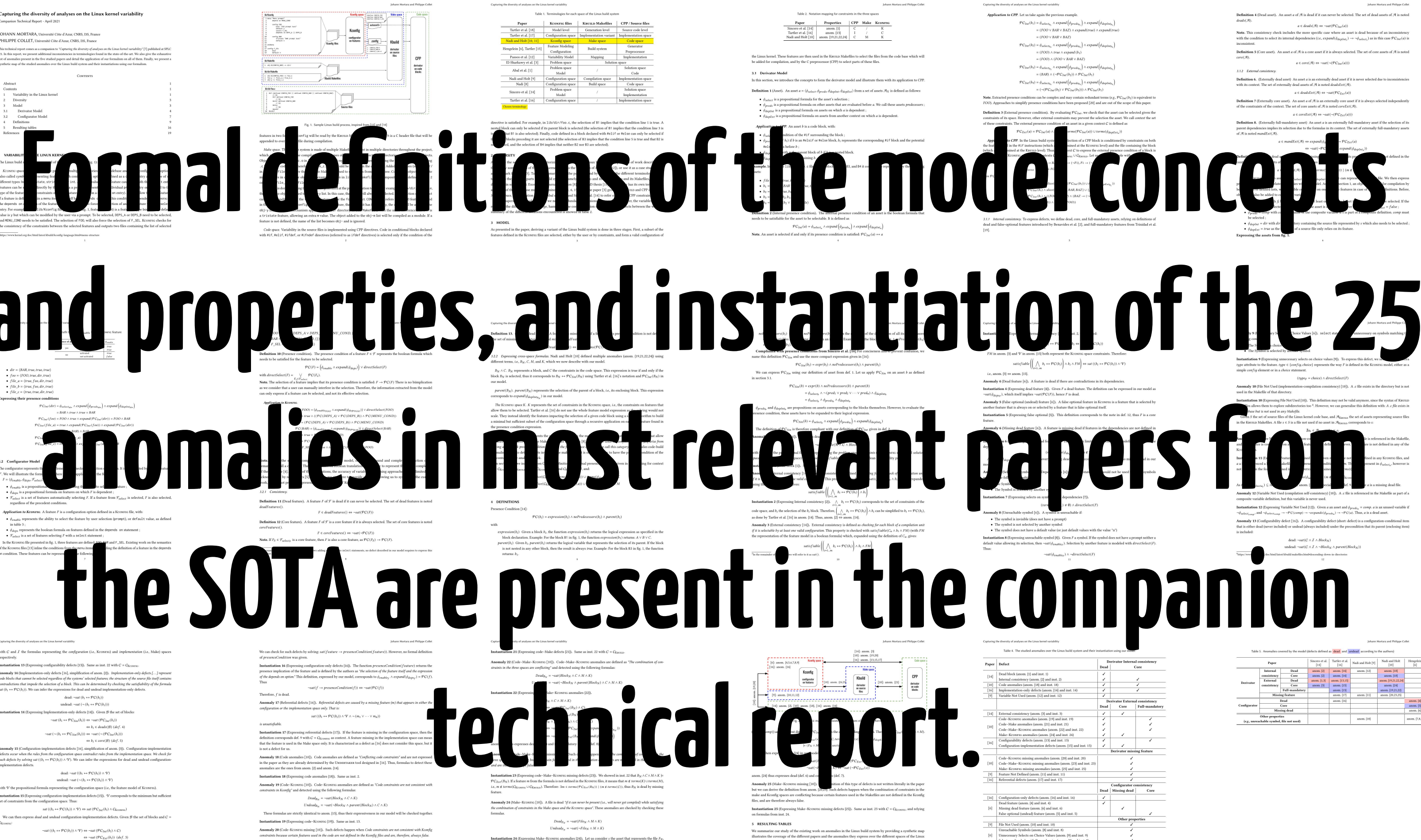
PC = Presence Condition

Contribution:



Internal PC (PC_{Int}) ⟶ Internal anomalies

External PC (PC_{Ext}) ⟶ External anomalies

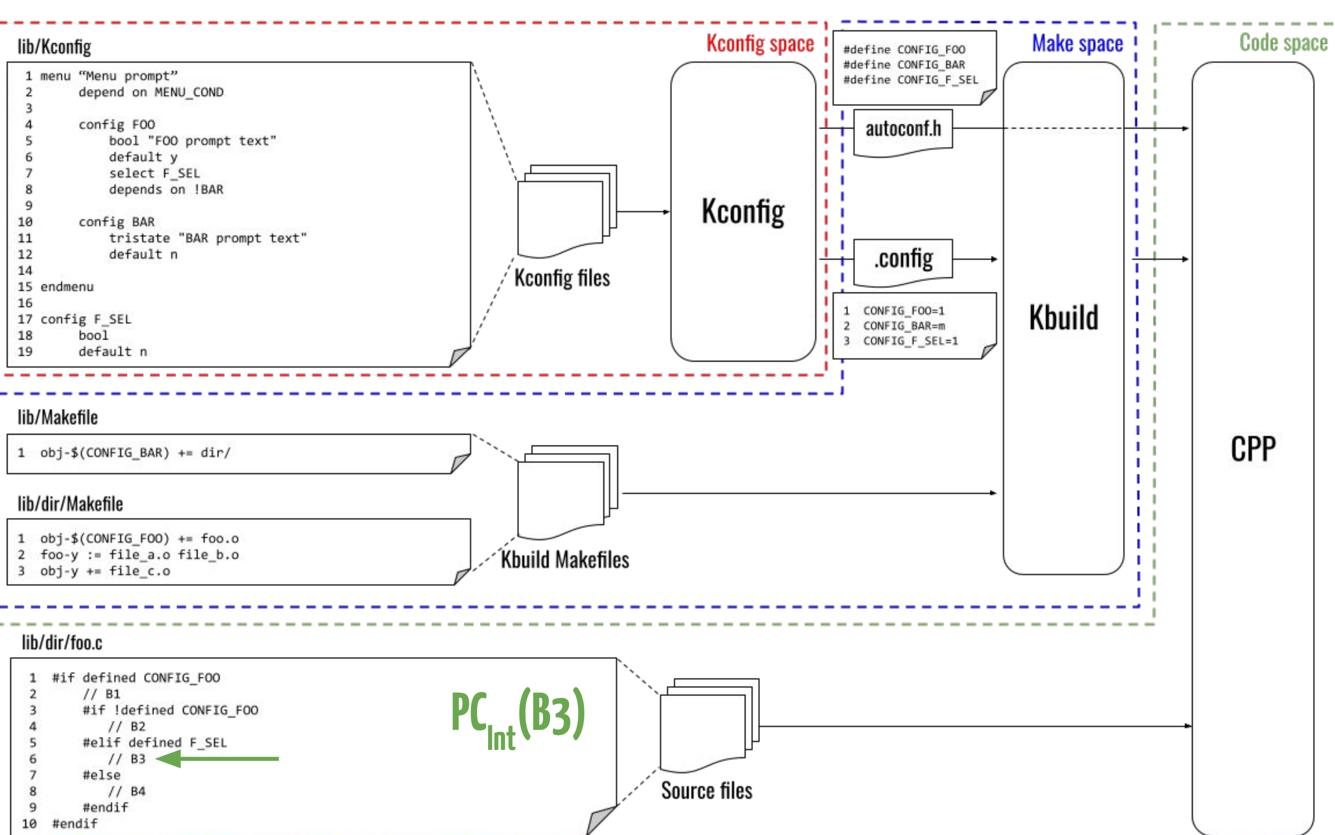


https://doi.org/10.5281/zenodo.4715969

How to build presence conditions?

We want to check if CPP block B3 is selectable.

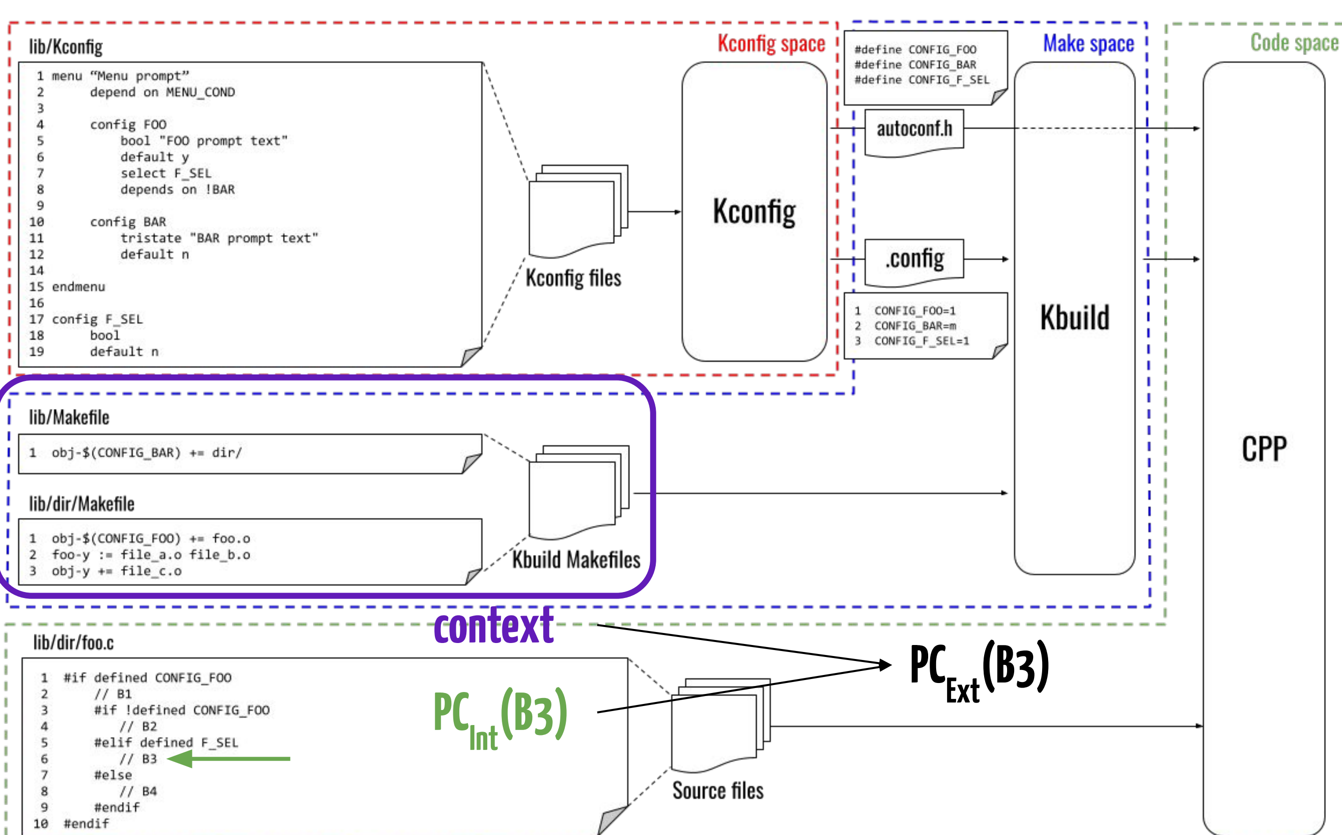
1. According to constraints **in its own space**, by determining $PC_{Int}(B3)$.



How to build presence conditions?

We want to check if CPP block B3 is selectable.

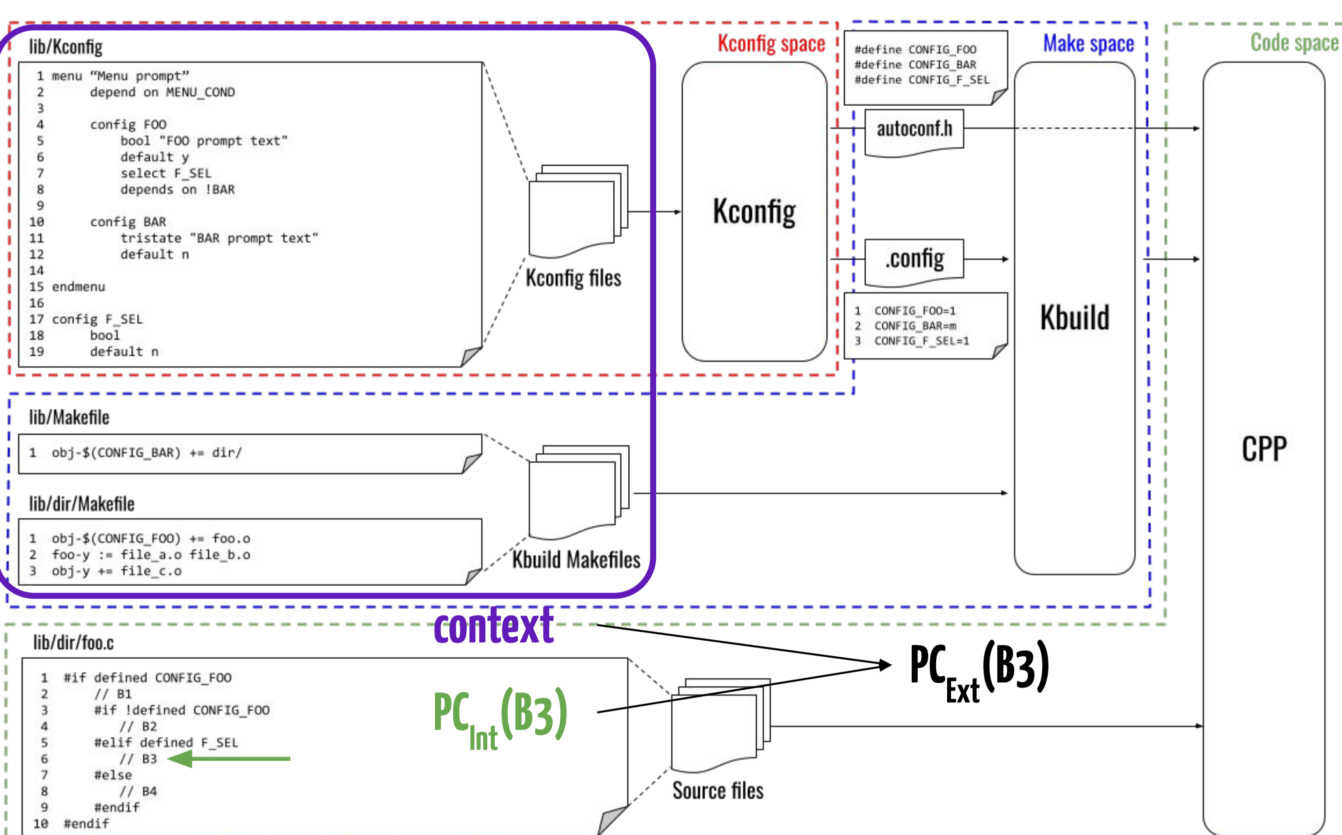
2. According to constraints **in its own space** and **in the Make space**, by determining $PC_{Ext}(B3)$ with the **Make space** as **context**.



How to build presence conditions?

We want to check if CPP block B3 is selectable.

3. According to constraints **in the three spaces**, by determining $PC_{Ext}(B3)$ with the **Kconfig** and **Make** spaces as **context**.



Step 1: building the internal presence condition

Code space

lib/dir/foo.c

```
1   #if defined CONFIG_FOO
2       // B1
3       #if !defined CONFIG_FOO
4         // B2
5       #elif defined F_SEL
6         // B3
7       #else
8         // B4
9       #endif
10  #endif
```

B3 is selectable if:

Step 1: building the internal presence condition

Code space

lib/dir/foo.c

```
1  #if defined CONFIG_FOO
2      // B1
3      #if !defined CONFIG_FOO
4        // B2
5      #elif defined F_SEL  ①
6        // B3
7      #else
8        // B4
9      #endif
10 #endif
```

B3 is selectable if:

1. its condition is satisfiable F_SEL

Step 1: building the internal presence condition

Code space

lib/dir/foo.c

```
1   #if defined CONFIG_FOO        ②
2       // B1
3       #if !defined CONFIG_FOO
4         // B2
5       #elif defined F_SEL       ①
6         // B3
7       #else
8         // B4
9       #endif
10  #endif
```

B3 is selectable if:

1. its condition is satisfiable F_SEL
2. its parent block (B1) is selectable $PC_{Int}(B1)$
   a. its condition is satisfiable, etc.

Step 1: building the internal presence condition

Code space

lib/dir/foo.c

```
1   #if defined CONFIG_FOO          (2)
2       // B1
3       #if !defined CONFIG_FOO     (3)
4         // B2
5       #elif defined F_SEL         (1)
6         // B3
7       #else
8         // B4
9       #endif
10  #endif
```

B3 is selectable if:

1. its condition is satisfiable F_SEL
2. its parent block (B1) is selectable $PC_{Int}(B1)$
   a. its condition is satisfiable, etc.
3. its predecessor B2 is not selectable $\neg PC_{Int}(B2)$

Step 1: building the internal presence condition

Code space

lib/dir/foo.c

```
1   #if defined CONFIG_FOO        ②
2       // B1
3       #if !defined CONFIG_FOO   ③
4         // B2
5       #elif defined F_SEL       ①
6         // B3
7       #else
8         // B4
9       #endif
10  #endif
```

B3 is selectable if:

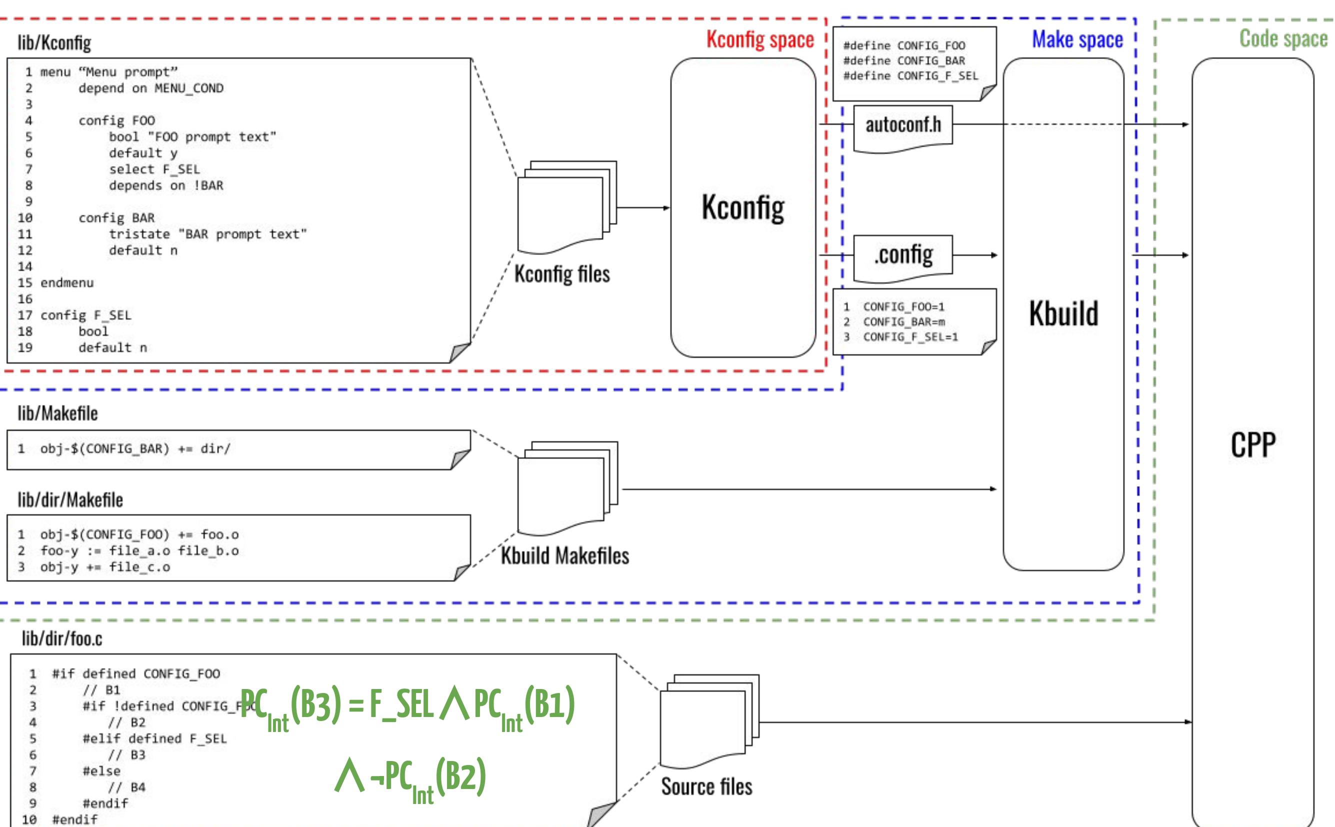
1. its condition is satisfiable F_SEL
2. its parent block (B1) is selectable $PC_{Int}(B1)$
   a. its condition is satisfiable, etc.
3. its predecessor B2 is not selectable $\neg PC_{Int}(B2)$

$$PC_{Int}(B3) = F_SEL \land PC_{Int}(B1) \land \neg PC_{Int}(B2)$$

Step 1: building the internal presence condition ✅

We want to check if CPP block B3 is selectable.

1. According to constraints **in its own space**, by determining **$PC_{Int}(B3)$**.



Step 2: adding constraints from the **Make space**

lib/Makefile

```
1  obj-$(CONFIG_BAR) += dir/
```

lib/dir/Makefile

```
1  obj-$(CONFIG_FOO) += foo.o
2  foo-y := file_a.o file_b.o
3  obj-y += file_c.o
```

To be selectable, B3's containing file (foo.c) needs to be selectable too!

Step 2: adding constraints from the **Make space**

lib/Makefile

```
1  obj-$(CONFIG_BAR) += dir/
```

lib/dir/Makefile

```
1  obj-$(CONFIG_FOO) += foo.o  ①
2  foo-y := file_a.o file_b.o
3  obj-y += file_c.o
```

To be selectable, B3's containing file (foo.c) needs to be selectable too!

1. foo.c's condition is satisfiable **FOO**

Step 2: adding constraints from the **Make space**

lib/Makefile

```
1  obj-$(CONFIG_BAR) += dir/
```
②

lib/dir/Makefile

```
1  obj-$(CONFIG_FOO) += foo.o
2  foo-y := file_a.o file_b.o
3  obj-y += file_c.o
```
①

To be selectable, B3's containing file (foo.c) needs to be selectable too!

1. foo.c's condition is satisfiable **FOO**
2. foo.c's parent directory is selectable

$PC_{Int}(dir)$

Step 2: adding constraints from the **Make space**

lib/Makefile

```
1  obj-$(CONFIG_BAR) += dir/
```

②

lib/dir/Makefile

```
1  obj-$(CONFIG_FOO) += foo.o
2  foo-y := file_a.o file_b.o
3  obj-y += file_c.o
```

①

$$PC_{Int}(foo.c) = FOO \wedge PC_{Int}(dir)$$

To be selectable, B3's containing file (foo.c) needs to be selectable too!

1. foo.c's condition is satisfiable FOO
2. foo.c's parent directory is selectable $PC_{Int}(dir)$

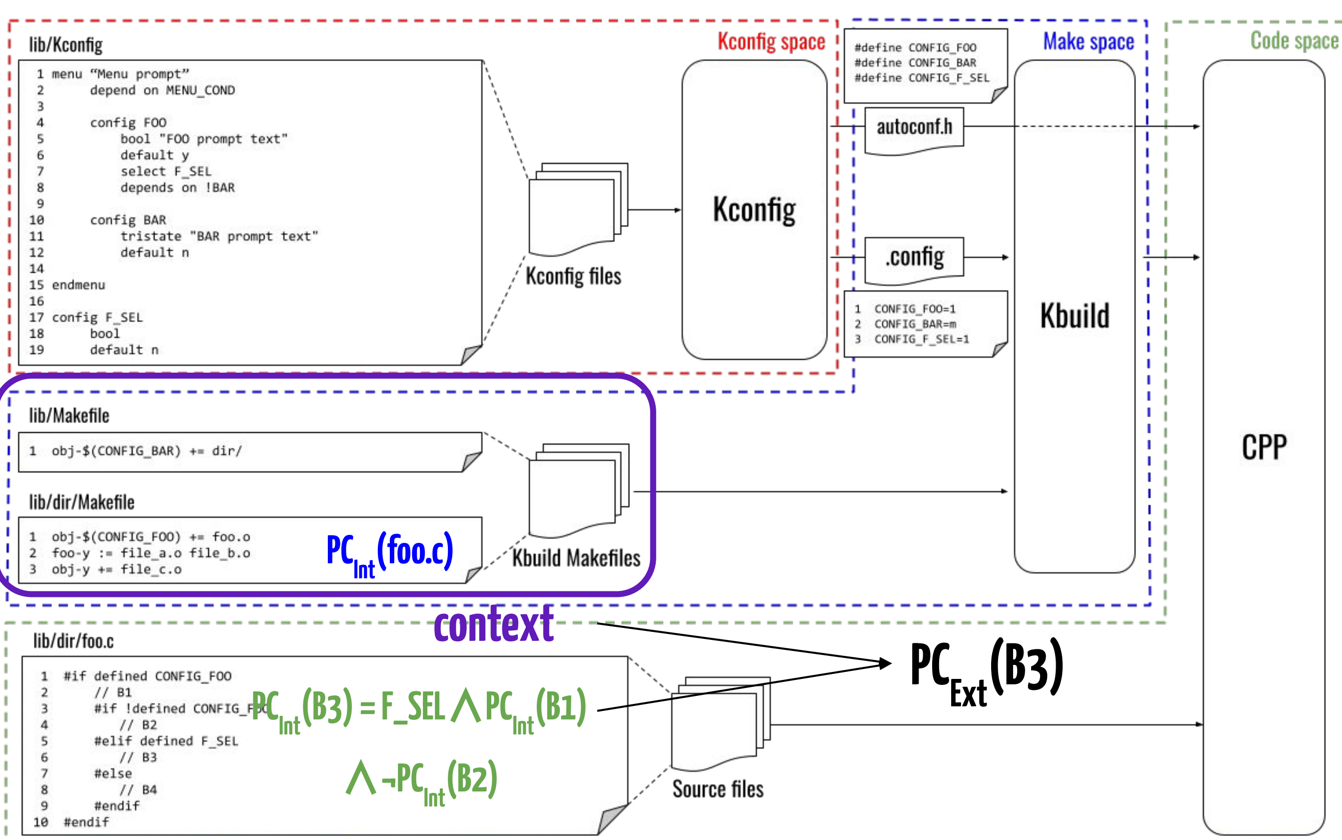
context

$$PC_{Ext}(B3) = PC_{Int}(B3) \wedge PC_{Int}(foo.c)$$

$PC_{Ext}(B3) = PC_{Int}(B3) \wedge PC_{Int}(foo.c)$

We want to check if CPP block B3 is selectable.

2. According to constraints **in its own space** and **in the Make space**, by determining $PC_{Ext}(B3)$ with the **Make space** as **context**.



Step 3: adding constraints from the Kconfig space

lib/Kconfig

```
1 menu “Menu prompt”
2       depends on MENU_COND
3
4       config FOO
5            bool "FOO prompt text"
6            default y
7            select F_SEL
8            depends on !BAR
9
10      config BAR
11           tristate "BAR prompt text"
12           default n
14
15 endmenu
16
17 config F_SEL
18      bool
19      default n
```

PCs for B3 and foo.c rely on features from the Kconfig space, so constraints between the features must also be satisfiable!

Step 2: adding constraints from the Kconfig space

lib/Kconfig

```
1 menu “Menu prompt”
2      depends on MENU_COND
3
4      config FOO
5          bool "FOO prompt text"
6          default y
7          select F_SEL
8          depends on !BAR
9
10     config BAR
11         tristate "BAR prompt text"
12         default n
14
15 endmenu
16
17 config F_SEL
18     bool
19     default n
```

①, ②, ③ (annotations next to the FOO, BAR and F_SEL blocks)

PCs for B3 and foo.c rely on features from the Kconfig space, so constraints between the features must also be satisfiable!

1. PC(FOO) = $\neg$BAR $\wedge$ MENU_COND
2. PC(BAR) = MENU_COND
3. PC(F_SEL) = true

Step 2: adding constraints from the Kconfig space

lib/Kconfig

```
1  menu “Menu prompt”
2       depends on MENU_COND
3
4       config FOO
5           bool "FOO prompt text"        ①
6           default y
7           select F_SEL
8           depends on !BAR
9
10      config BAR
11          tristate "BAR prompt text"    ②
12          default n
14
15 endmenu
16
17 config F_SEL
18      bool                              ③
19      default n
```

PCs for B3 and foo.c rely on features from the Kconfig space, so constraints between the features must also be satisfiable!

1. $PC(FOO) = \neg BAR \wedge MENU_COND$
2. $PC(BAR) = MENU_COND$
3. $PC(F_SEL) = true$

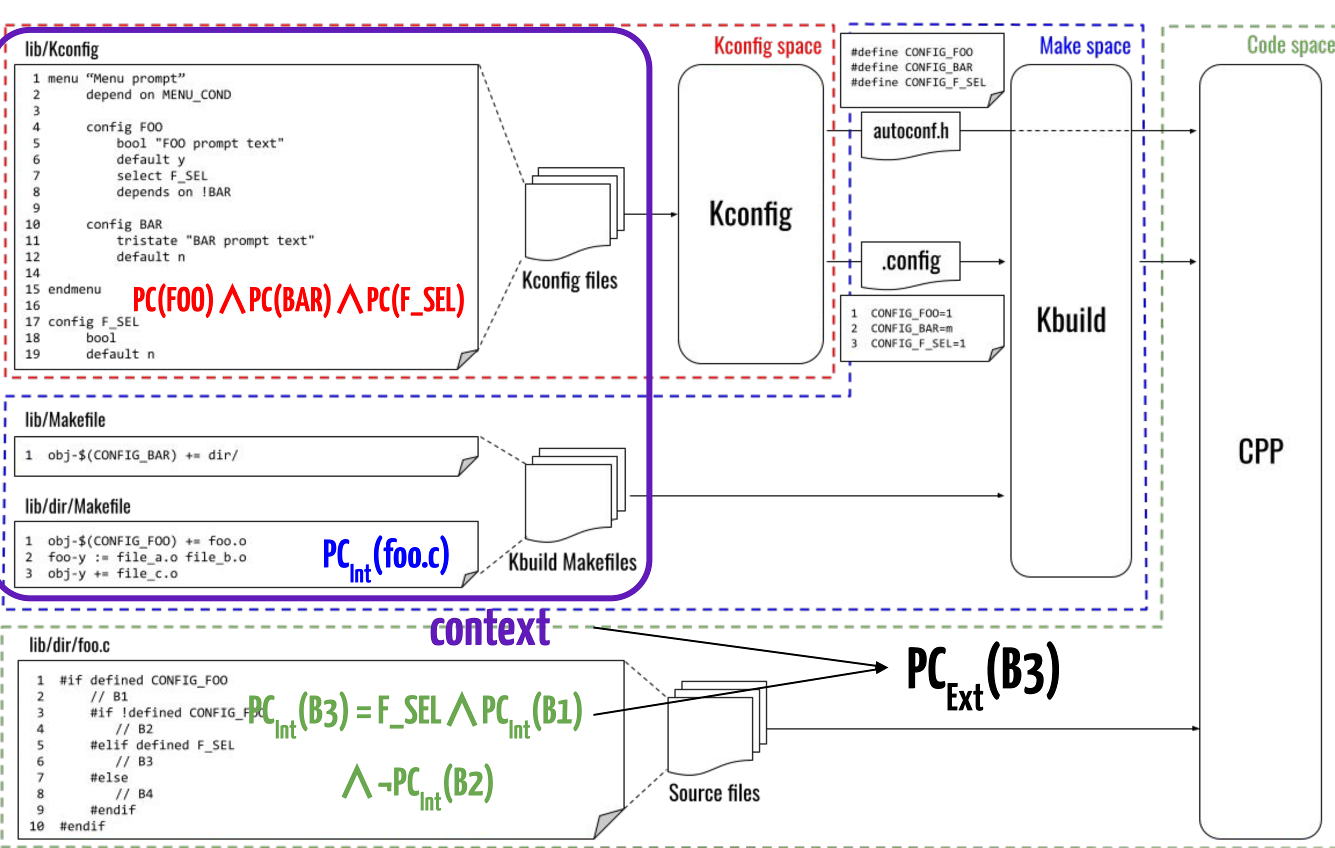
context

$$PC_{Ext}(B3) = PC_{Int}(B3) \wedge \underbrace{PC_{Int}(foo.c) \wedge PC(FOO) \wedge PC(BAR) \wedge PC(F_SEL)}_{context}$$

$PC_{Ext}(B3) = PC_{Int}(B3) \wedge PC_{Int}(foo.c) \wedge PC(FOO) \wedge PC(BAR) \wedge PC(F_SEL)$

We want to check if CPP block B3 is selectable.

3. According to constraints **in the three spaces**, by determining $PC_{Ext}(B3)$ with the **Kconfig** and **Make spaces** as **context**.



Anomaly	SOTA	Our model
Internal dead block	$\neg sat(B3 \land C)$	$\neg sat(PC_{Int}(B3)) = \neg sat(F_SEL \land PC_{Int}(B1) \land \neg PC_{Int}(B2))$
External dead block, Make as context	$\neg sat(B3 \land C \land M)$	$\neg sat(PC_{Ext}(B3)) = \neg sat(PC_{Int}(B3) \land PC_{Int}(foo.c))$
External dead block, Make and Kconfig as context	$\neg sat(B3 \land C \land M \land K)$	$\neg sat(PC_{Ext}(B3)) = \neg sat(PC_{Int}(B3) \land PC_{Int}(foo.c) \land PC(FOO) \land PC(BAR) \land PC(F_SEL))$

Results coverage

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}
	Mandatory						anom. {5}
	Missing dead						anom. {6}
Other properties (*e.g.*, unreachable symbol, file not used)					anom. {10}		anom. {7,8,9}

All anomalies could be expressed in our unified model

Results coverage

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}
	Mandatory						anom. {5}
	Missing dead						anom. {6}
Other properties (*e.g.*, unreachable symbol, file not used)					anom. {10}		anom. {7,8,9}

All anomalies could be expressed in our unified model

Anomalies with identical names check different defects

Results coverage

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}
	Mandatory						anom. {5}
	Missing dead						anom. {6}
Other properties (*e.g.*, unreachable symbol, file not used)					anom. {10}		anom. {7,8,9}

All anomalies could be expressed in our unified model

Anomalies with identical names check different defects

Anomalies with different names check identical defects

Future work

Provide a model-driven framework for the proposed model

Apply the model to the build systems of other systems:

- BusyBox
- JHipster
- Mozilla Firefox

Capturing the diversity of analyses on the Linux kernel variability

Johann Mortara – Philippe Collet

Map of the existing work on anomalies in the Linux build system

Unified model to represent anomalies in the Linux kernel build system

Covering SOTA properties and **exhibiting incoherences** between them

Get the paper:

https://doi.org/10.1145/3461001.3471151

Get the technical report:

https://doi.org/10.5281/zenodo.4715969