Formal Semantics of the Kconfig Language

Technical Note

Steven She[1] and Thorsten Berger[2]

[1]shshe@gsd.uwaterloo.ca, University of Waterloo, Canada
[2]berger@informatik.uni-leipzig.de, University of Leipzig, Germany

January 2010

Abstract

The Kconfig language defines a set of symbols that are assigned a value in a configuration. We describe the semantics of the Kconfig language according to the behaviour exhibited in the xconfig configurator. We assume an abstract syntax representation for concepts in the Kconfig language and delegate the details of the translation from concrete to abstract syntaxes to a later document.

1 Abstract syntax

Identifiers and expressions. We start be defining the preliminary concepts available in the Kconfig language. Let Id be a finite set of names identifying a symbol—more precisely, $\mathsf{Id} \in \mathcal{P}(\mathsf{String})$. Let $\mathsf{Const} = \mathsf{Tri} \cup \mathsf{String} \cup \mathsf{Hex} \cup \mathsf{Int}$ be the set of values assignable to each feature and available as constants in expressions, where $\mathsf{Tri} = \{0_\mathsf{t}, 1_\mathsf{t}, 2_\mathsf{t}\}$. Tri is ordered such that $0_\mathsf{t} < 1_\mathsf{t} < 2_\mathsf{t}$. The Tri, String, Hex, and Int domains are disjoint (i.e. mutually exclusive). We can now define an expression in the Kconfig language. $\mathit{KExpr}(\mathsf{Id})$ is a set of expressions over Id generated by the following grammar, where $e \in \mathit{KExpr}(\mathsf{Id})$, $\mathit{iv} \in \mathsf{Id} \cup \mathsf{Const}$, $\otimes \in \{\text{or}, \text{and}\}$, $\ominus \in \{=, \neq\}$:

$$e ::= e \otimes e \mid \text{not}\, e \mid \mathit{iv} \ominus \mathit{iv} \mid \mathit{iv} \tag{1}$$

Evaluating a *KExpr* returns a tristate value (i.e. $v \in \mathsf{Tri}$). We will define the semantics of an *eval* function in Section 2.2.

Kconfig model. $\mathsf{Kconfig}$ denotes the set of all possible models in the Kconfig language. Thus a single Kconfig model $m \in \mathsf{Kconfig}$ is a tuple consisting of a set of *configs* and a set of *choices*. $\mathsf{Kconfig}$ is defined as:

$$\mathsf{Kconfig} = \mathcal{P}(\mathsf{Configs}) \times \mathcal{P}(\mathsf{Choices}) \tag{2}$$

Given a Kconfig model $m \in \mathsf{Kconfig}$, we define the shorthand m_config to refer to its set of configs and m_choice to refer to its set of choices.

[0]The semantics defined in this document directly reflect the behavior of the Linux *make xconfig* tool, which could — in some specific cases — act differently from what the Kconfig language developers originally had in mind. At least in case of the *reverse dependency* the documentation explicitly states the following gap: "*Select* should be used with care. *Select* will force a symbol to a value without visiting the dependencies. By abusing *select* you are able to select a symbol FOO even if FOO depends on BAR that is not set."

Configs are the primary components of a Kconfig model. A config defines a unique identifier with type, a prompt condition — a condition that determines when a config becomes user-changeable, a list of defaults, a expression denoting its reverse dependency — the conditions that would forcefully enable this feature through a *select* statement, and a set of ranges — restrictions on the value for configs or hex type. We define $\mathsf{Configs}$as follows:

$$\mathsf{Configs} = \mathsf{Id} \times \mathsf{Type} \times \mathit{KExpr}(\mathsf{Id}) \times \mathsf{Default} * \times \mathit{KExpr}(\mathsf{Id}) \times \mathcal{P}(\mathsf{Range}) \tag{3}$$

where,

- $\mathsf{Type} = \{\text{boolean}, \text{tristate}, \text{int}, \text{hex}, \text{string}\}$ denotes a type and consequently the possible values for the config.
- $\mathsf{Default} = \mathit{KExpr}(\mathsf{Id}) \times \mathit{KExpr}(\mathsf{Id})$ denotes defaults. The first KExprdenotes a default expression (i.e. that is evaluated and assigned to the symbol) and the second KExprdenotes the condition required for the default to become effective.
- $\mathsf{Range} = (\mathsf{Int} \cup \mathsf{Hex} \cup \mathsf{Id}) \times (\mathsf{Int} \cup \mathsf{Hex} \cup \mathsf{Id}) \times \mathit{KExpr}(\mathsf{Id})$ is a triple consisting of a lower bound, an upper bound and a condition. Note the absence of Tri in the lower and upper bounds of the range; this is due to ranges being only effective on int and hex-typed configs as we will describe in the following paragraph.

We further define a function $\mathsf{Id}(m)$ to denote identifiers of configs in the model m:

$$\mathsf{Id}(m) = \{n \mid (n, \_, \_, \_, \_, \_) \in m_{\mathsf{config}}\} \tag{4}$$

The second component of $\mathsf{Kconfig}$ refers to a set of choice nodes. A choice is an abstract construct that defines no symbol in the configuration, however, it imposes additional constraints on its nested elements. We define choices as a quadruple consisting of a type where boolean or tristate are the only valid types, a flag indicating whether the choice is mandatory, a prompt condition followed by a set of identifiers indicating its members. The set $\mathsf{Choices}$ is defined as:

$$\mathsf{Choices} = \{\text{boolean}, \text{tristate}\} \times \mathsf{Bool} \times \mathit{KExpr}(\mathsf{Id}) \times \mathcal{P}(\mathsf{Id}(m)) \tag{5}$$

Well-formedness rules. Given an element $(\_, t, \_, \_, rev, rngs) \in \mathsf{Configs}$, this config is well-formed if the following conditions are satisfied:

- The reverse dependency of configs with type int, hex or string must be 0_{t}. In other words, no config may select a config that is not of type boolean or tristate.

$$(rev \neq 0_{\mathsf{t}}) \implies t = \text{boolean} \vee t = \text{tristate} \tag{6}$$

- Ranges can only be defined on configs with a numerical type, namely int or hex types. Thus, the following constraint must hold for a config to be well-formed:

$$(|rngs| > 0) \implies t = \text{int} \vee t = \text{hex} \tag{7}$$

Brief note on concrete syntax translation. *Menuconfigs* and *menus* are first-class concepts in the concrete syntax of the Kconfig language. However, both of these concepts are not present in the abstract syntax. First, menuconfigs are semantically identical to configs and only differ in terms of its appearance in the configurator; thus, we model menuconfigs as configs in the abstract syntax. Menus do not define a symbol; thus menus are not present in a configuration. However, menus can impose constraints on its nested elements. We handle these constraints via a syntactic rewrite on the prompt, default and range conditions of all nested symbols. Details for this syntactic rewrite will be provided in later document.

2 Semantics

2.1 Semantic domain

A configuration of a $\mathsf{Kconfig}$ model is an assignment of values $v \in \mathsf{Const}$ to config elements. Thus, the set of all possible configurations is defined as:

$$\mathsf{Confs} = \mathsf{Id} \to \lfloor \mathsf{Const} \rfloor \tag{8}$$

If $c \in \mathsf{Confs}$ and $x \in \mathsf{Id}$, we write $c(x)$ in order to refer to the value of identifier x under the configuration c. Now, we define the semantics of a $\mathsf{Kconfig}$ model in terms of sets of configurations. Thus, $\mathcal{P}(\mathsf{Confs})$ is our semantic domain. We define $[\![\cdot]\!]_{\mathsf{kconfig}}$ as the function that evaluates a Kconfig model and returns a set of valid configurations:

$$[\![\cdot]\!]_{\mathsf{kconfig}} \colon \mathsf{Kconfig} \to \mathcal{P}(\mathsf{Confs}) \tag{9}$$

2.2 Global functions

We start with the definition of some functions used throughout the semantics. First, we define an interpretation of tristate values in boolean logic with bool: $\mathsf{Tri} \to \mathsf{Bool}$ where $\mathsf{Bool} = \{T, F\}$:

$$\mathrm{bool}(v) = \begin{cases} F & \text{iff } v = 0_t \\ T & \text{iff } v = 1_t \lor v = 2_t \end{cases} \tag{10}$$

Moreover, we define a function *access* : $(\mathsf{Id} \cup \mathsf{Const}) \times \mathsf{Confs} \to \mathsf{Const}$ that retrieves the value of either a constant or a symbol. When an identifier has the value of $\perp$ (to be defined in Equation 14), then the *access* function returns the identifier itself in the form of a string:

$$\mathrm{access}(iv, c) = \begin{cases} iv & \text{iff } iv \in \mathsf{Const} \lor (iv \in \mathsf{Id} \land c(iv) = \perp) \\ c(iv) & \text{otherwise} \end{cases} \tag{11}$$

Next, we define the function *toStr*: $\mathsf{Const} \to \mathsf{String}$ that models the translation of a constant to a string representation. Let $i \in \mathsf{Int}$, $h \in \mathsf{Hex}$ and $s \in \mathsf{String}$, in the following definition of *toStr*:

$$\begin{aligned} &\mathit{toStr}(0_\mathsf{t}) = \text{“n”} \quad &\mathit{toStr}(1_\mathsf{t}) = \text{“m”} \quad &\mathit{toStr}(2_\mathsf{t}) = \text{“y”} \\ &\mathit{toStr}(i) = \text{“”} + i \quad &\mathit{toStr}(h) = \text{“0x”} + h \quad &\mathit{toStr}(s) = s \end{aligned} \tag{12}$$

where the + operator is string concatenation.

Finally, the function *eval*: $\mathit{KExpr}(\mathsf{Id}) \to \mathsf{Tri}$ describes the evaluation of a KExpr in the Kconfig language. We define *eval* recursively with $e_1, e_2 \in \mathit{KExpr}(\mathsf{Id})$ and $iv, iv_x, iv_y \in \mathsf{Id} \cup \mathsf{Const}$:

$$\begin{aligned} \mathit{eval}(iv_x = iv_y, c) &= \begin{cases} 2_\mathsf{t} & \text{iff } \mathit{toStr}(\mathrm{access}(iv_x, c)) = \mathit{toStr}(\mathrm{access}(iv_y, c)) \\ 0_\mathsf{t} & \text{otherwise} \end{cases} \\ \mathit{eval}(iv_x \neq iv_y, c) &= 2_\mathsf{t} - \mathit{eval}(iv_x = iv_y, c) \\ \mathit{eval}(\mathrm{not}\, e_1, c) &= 2_\mathsf{t} - \mathit{eval}(e_1, c) \\ \mathit{eval}(e_1 \,\mathrm{and}\, e_2, c) &= \min(\mathit{eval}(e_1, c), \mathit{eval}(e_2, c)) \\ \mathit{eval}(e_1 \,\mathrm{or}\, e_2, c) &= \max(\mathit{eval}(e_1, c), \mathit{eval}(e_2, c)) \\ \mathit{eval}(iv, c) &= \begin{cases} v_{iv} & \text{iff } v_{iv} = \mathrm{access}(iv, c) \land v_{iv} \in \mathsf{Tri} \\ 0_\mathsf{t} & \text{otherwise} \end{cases} \end{aligned} \tag{13}$$

2.3 Valuation functions

Kconfig model. We begin by defining the $[\![\cdot]\!]_{\mathsf{kconfig}}$. Given a Kconfig model $m \in \mathsf{Kconfig}$, the semantics of a model is the intersection of all denotations across the model, configs and choices. In other words, the set of valid configurations for a Kconfig model is those configurations that satisfy all denotations. $[\![\cdot]\!]_{\mathsf{kconfig}} \colon \mathsf{Kconfig} \to \mathsf{Confs}$ is defined:

$$\begin{aligned} [\![m]\!]_{\mathsf{kconfig}} = \left(\bigcap_{n \in m_{\mathsf{config}}} [\![n]\!]_{\mathsf{type}} \cap [\![n]\!]_{\mathsf{bounds}} \cap [\![n]\!]_{\mathsf{default}} \cap [\![n]\!]_{\mathsf{range}} \right) \cap \left(\bigcap_{n \in m_{\mathsf{choice}}} [\![n]\!]_{\mathsf{choice}} \right) \\ \cap [\![m]\!]_{\mathsf{module}} \\ \cap [\![m]\!]_{\mathsf{undeclared}} \end{aligned} \tag{14}$$

Type. The first denotation pertains to the constraints imposed by a config's type. The type of a config restricts its valid values to those in its respective domain. $[\![\cdot]\!]_{\mathsf{type}} \colon \mathsf{Configs} \to \mathsf{Confs}$ is defined:

$$[\![(n, t, \_, \_, \_, \_)]\!]_{\mathsf{type}} = \begin{cases} \{c \in \mathsf{Confs} \mid c(n) \in \mathsf{Tri} \setminus \{1_{\mathsf{t}}\}\} & \text{iff } t = \text{boolean} \\ \{c \in \mathsf{Confs} \mid c(n) \in \mathsf{Tri}\} & \text{iff } t = \text{tristate} \\ \{c \in \mathsf{Confs} \mid c(n) \in \mathsf{String}\} & \text{iff } t = \text{string} \\ \{c \in \mathsf{Confs} \mid c(n) \in \mathsf{Hex} \cup \{\text{``''}\}\} & \text{iff } t = \text{hex} \\ \{c \in \mathsf{Confs} \mid c(n) \in \mathsf{Int} \cup \{\text{``''}\}\} & \text{iff } t = \text{int} \end{cases} \tag{15}$$

Upper and lower bounds. Next, the bounds denotation models the lower and upper bounds of a config. The lower bound is determined by the evaluation of a config's reverse dependency. Recall that the reverse dependency models the behaviour of the *select* statement in the concrete syntax. The upper bound is defined by a config's prompt condition. This denotation has no effect on configs of type int, hex, or string since the the reverse dependency that determines a lower bound is 0_{t} by our well-formedness rules, and the *eval* function returns 0_{t} when evaluating a value not in Tri. $[\![\cdot]\!]_{\mathsf{bounds}} \colon \mathsf{Configs} \to \mathsf{Confs}$ is defined:

$$\begin{aligned} &[\![(n, \_, pro, \_, rev, \_)]\!]_{\mathsf{bounds}} = \\ &\qquad \{c \in \mathsf{Confs} \mid eval(c(n), c) \geq \mathsf{Lower}(c) \wedge (\mathsf{Upper}(c) < \mathsf{Lower}(c) \vee eval(c(n), c) \leq \mathsf{Upper})\} \end{aligned} \tag{16}$$

where $\mathsf{Lower}(\mathrm{c}) = eval(\mathrm{rev,c})$ and $\mathsf{Upper}(\mathrm{c}) = eval(\mathrm{pro,c})$.

Defaults. Kconfig has support for setting a default expression for a config. The default expression interacts with the prompt condition that determines when the config is user-changeable. When the prompt condition is satisfied, then the user is free to set a value. However, when the prompt condition is not satisfied, the default determine the config's value. $[\![\cdot]\!]_{\mathsf{default}} \colon \mathsf{Configs} \to \mathsf{Confs}$ is defined:

$$\begin{aligned} &[\![(n, \_, \_, defs, rev, \_)]\!]_{\mathsf{default}} = \\ &\qquad \{c \in \mathsf{Confs} \mid \mathrm{bool}(eval(pro, c)) \vee c(n) = \max(eval(\mathrm{default}(defs, c)), eval(rev, c))\} \end{aligned} \tag{17}$$

where $default \colon \mathcal{P}(\mathsf{Default}) \times \mathsf{Type} \times \mathsf{Confs} \to \mathsf{Const}$ is a function that models the retrieval of a default. Recall that $defs$ is a list of defaults (and thus ordered). The effect of a default's value depends on the type of its defining config. If the config is *boolean* or *tristate*, then the default value is evaluated to a value in Tri. Otherwise, the default value must be either an element of Const or Id. Let Nil be the empty list and :: be the list *cons* operator. Let $t_{\mathsf{Tri}} \in \{\text{boolean}, \text{tristate}\}$ and

$t_{\mathsf{Entry}} \in \{\text{int}, \text{hex}, \text{string}\}$. The *default* function is defined recursively, so we begin by defining its base cases:

$$\begin{aligned} \text{default}(Nil, t_{\mathsf{Tri}}, c) &= 0_{\mathsf{t}} \\ \text{default}(Nil, t_{\mathsf{Entry}}, c) &= \text{``''} \end{aligned} \tag{18}$$

Equation 18 states that given an empty list of defaults, we return 0_{t} if the type is either boolean or tristate, or the empty string for types int, hex or string. Next, we define the recursive rule. In the following equation, we decompose the list into its head and tail components. First, we describe the function for boolean and tristate type:

$$\text{default}(\,(e, cond) :: rest, t_{\mathsf{Tri}}, c) = \begin{cases} eval(e, c) & \text{if bool}(eval(cond, c)) \\ \text{default}(rest, t_{\mathsf{Tri}}, c) & \text{otherwise} \end{cases} \tag{19}$$

Now for the remaining types:

$$\text{default}(\,(e, cond) :: rest, t_{\mathsf{Entry}}, c) = \begin{cases} \text{access}(e, c) & \text{if bool}(eval(cond, c)) \\ \text{default}(rest, t_{\mathsf{Entry}}, c) & \text{otherwise} \end{cases} \tag{20}$$

Ranges. Ranges impose a lower and upper bound on the value of int or hex configs. $[\![\cdot]\!]_{\mathsf{range}} : \mathsf{Configs} \rightarrow \mathsf{Confs}$ is defined as:

$$\begin{aligned} [\![n, \_, \_, \_, \_, rngs)]\!]_{\mathsf{range}} = \{c \in \mathsf{Confs} \mid\ & \forall (l, u, cond) \in rngs. \\ & \text{bool}(eval(cond, c)) \rightarrow c(n) \geq \text{access}(l, c) \wedge c(n) \leq \text{access}(u, c)\} \end{aligned} \tag{21}$$

Choices. A choice restricts the number of members that can be selected (i.e. have a value greater than 0_{t}). The choice denotation, $[\![\cdot]\!]_{\mathsf{choice}} : \mathsf{Choices} \rightarrow \mathsf{Confs}$ is defined:

$$\begin{aligned} & [\![(boolOrTri, isMand, prompt, mems)]\!]_{\mathsf{choice}} = \\ & \qquad \{c \in \mathsf{Confs} \mid\ bool(eval(prompt, c)) \rightarrow \mathsf{Xor} \wedge \mathsf{BChoice} \wedge \mathsf{Mandatory}\} \end{aligned} \tag{22}$$

where Xor defines the condition that one and only one member may be set to 2_{t}:

$$\mathsf{Xor} = \exists m_1 \in mems.\ (m_1 = 2_{\mathsf{t}}) \rightarrow (\forall m_2 \in mems \setminus \{m_1\}.\ m_2 = 0_{\mathsf{t}}) \tag{23}$$

If the choice is a boolean choice, then the only valid value for its members is 2_t. In combination with Xor, this defines that a boolean choice may have at most one member with a value not equal to 0_{t} and that member must be set to 2_{t}:

$$\mathsf{BChoice} = (boolOrTri = \text{boolean}) \rightarrow \exists m \in mems.\ c(m) = 2_{\mathsf{t}} \tag{24}$$

Finally, if the choice is *mandatory*, then at least one member must be selected:

$$\mathsf{Mandatory} = isMand \rightarrow \exists m \in mems.\ c(m) > 0_{\mathsf{t}} \tag{25}$$

Modules. A special MODULES config is used to specify support for modules in the kernel. Disabling MODULES disallows the 1_t state for configs and effectively turns all tristate configs into boolean configs. A special symbol m is used in expressions to identify a dependency on the MODULES feature in the concrete syntax. Configs with a dependency on m cannot be selected (i.e. must be set to 0_t) if MODULES is not selected. We assume that the special m identifier has been expanded to MODULES in the abstract syntax.

$$[\![m]\!]_{\mathsf{module}} = \{c \in \mathsf{Confs} \mid c(\text{MODULES}) = \text{n} \rightarrow \forall i \in \mathsf{Id}.\ c(i) \neq 1_t\} \tag{26}$$

	type	interpretation in tristate logic
X	tristate	$X = y$ or $X = m$
$\neg X$	tristate	$X = n$
X	boolean	$X = y$
$\neg X$	boolean	$X = n$
X	string	$X =$ "..." (some non-empty string)
$\neg X$	string	$X =$ ""
X	int	$X = i$ (some integer, including 0)
$\neg X$	int	$X =$ ""
X	hex	$X = i$ (some hex)
$\neg X$	hex	$X =$ ""

Table 1: Interpretation of propositional variables

Undeclared symbols. We also define the behaviour of *undeclared symbols.* The Kconfig language supports references to symbols that are not declared in constraints. These undeclared symbols are assigned the special symbol $\perp$ in our semantics. The use of this symbol will become apparent in the definition of the *eval* function in Section 2.2. The $[\![\cdot]\!]_{\mathsf{undeclared}} \colon \mathsf{Kconfig} \to \mathcal{P}(\mathsf{Confs})$ denotation is defined as:

$$[\![m]\!]_{\mathsf{undeclared}} = \{c \in \mathsf{Confs} \mid \forall x \in \mathsf{Id} \setminus \mathsf{Id}(m).\ c(x) = \perp\} \tag{27}$$

3 1-Var Propositional Semantics

The goal of the propositional semantics is to achieve a weakening of the constraints of the full semantics.

Rewrite rules for expressions. The *rewrite* is a partial function that implements rewrite rules on expressions. The function $rewrite \colon KExpr(\mathsf{Id}) \to KExpr(\mathsf{Id})$ is defined as:

$$rewrite(e) = \begin{cases} 0 & \text{if } (e \text{ is an variable } \wedge typeOf(e) \in \{int, hex, string\}) \vee e = 0_t \\ 1 & \text{if } e = 1_t \vee e = 2_t \\ X \leftrightarrow Y & \text{if } e \text{ is } X = Y \text{ where X and Y are variables} \\ \textit{Use Table 1} & \text{if } e \text{ is } X = lit \vee X \neq lit \end{cases} \tag{28}$$

We further define the function $relax \colon KExpr(\mathsf{Id}) \to KExpr(\mathsf{Id})$ which converts an expression to CNF and removes clauses equivalent to equality checks. This function is used to relax the constraints on the antecedent (LHS) of an implication.

Semantics. In the propositional semantics, we model the set of propositional configurations as Confs_p:

$$\mathsf{Confs}_p = \mathsf{Id} \to \mathsf{Bool} \tag{29}$$

The $default$ function is defined as $\mathsf{Id} \times \mathsf{Default}* \times \mathsf{Confs}_p \to \mathsf{Bool}$. An extra parameter providing the declaring identifier is needed for the propositional semantics.

$$default(n, defs, c) = \begin{cases} \neg n & \text{if no default conditions are satisfied} \\ rewrite(n = eval(iv_i, c)) & \text{otherwise if } t \in \{boolean, tristate\}, \\ & \text{where } iv_i \text{ is the 1st matching default value} \\ n & \text{otherwise if } t \in \{int, hex, string\} \end{cases} \tag{30}$$

The default denotation is defined as:

$$[\![(n, t, vis, pro, defs, rev, rngs)]\!]_{\mathsf{default}} = \{c \in \mathsf{Confs}_p \mid eval(pro, c) \lor default(n, defs, c)\} \tag{31}$$

The constraint denotation which models constraints imposed by the reverse dependency and visibility conditions is defined as:

$$[\![(n, t, vis, pro, defs, rev, rngs)]\!]_{\mathsf{bounds}} = \{c \in \mathsf{Confs}_p \mid (eval(relax(rev), c) \to c(n)) \land (c(n) \to eval(vis, c))\} \tag{32}$$

We ignore ranges since we abstract away the value of each config. We also assume that the MODULE config is enabled, thus allowing for the 1_t state in tristate configs.

$$[\![(boolOrTri, isMand, vis, mems)]\!]_{\mathsf{choice}} = \left\{ c \in \mathsf{Confs} \mid eval(vis, c) \to choose(1, ids(mems)) \land \left(isMand \to \bigvee_{m \in mems} m \right) \right\} \tag{33}$$