**Kconfig语言的形式语义学**

Steven She1 and Thorsten Berger2

1shshe@gsd.uwaterloo.ca， 加拿大，滑铁卢大学

2berger@informatik.uni-leipzig.de， 德国，莱比锡城大学

2010年1月

**摘 要**

Kconfig语言定义了一组在配置中被赋值的符号。我们依据xconfig配置程序表现出的行为来描述Kconfig语言的语义。我们为Kconfig语言中的概念假设一个抽象的语法陈述，并且将由具体到到抽象的语法转化的细节总结成文档。

**1.抽象语法**

**标识符和表达式**. 我们开始定义在Kconfig语言中可利用的预备概念。 假设 Id 为一个识别符号名的有限集合——更确切地说，Id ∈ P(String)。假设 Const = Tri ∪ String ∪ Hex ∪ Int 为一个对各个特征和变量(如表达式中的常量)可赋值的值的集合。Tri = {0t, 1t, 2t}， Tri 是有序的，0t < 1t < 2t.Tri, String, Hex, 和 Int 的域是不连接的(即互斥的)。我们现在可以定义一个Kconfig语言的表达式。KExpr(Id) 是 Id 依据下面的语法生成的表达式的一个集合 , e ∈ KExpr(Id), iv ∈ Id ∪ Const, ⊗ ∈ {or, and}, ⊖ ∈ {=, = 6 }:

e ::= e ⊗ e | not e | iv ⊖ iv | iv (1)

Evaluating a KExpr 返回一个三态值 (即 v ∈ Tri)。 我们会在2.2节定义估值函数的语义。

**Kconfig模式**. Kconfig 在 Kconfig 语言中表示所有可能模式的集合。因此一个单 Kconfig 模型 m ∈ Kconfig 是一个 configs 和 choices 集合构成的元组。

Kconfig 定义为:

Kconfig = P(Configs) × P(Choices) (2)

给定一个 Kconfig 模型 m ∈ Kconfig，我们定义 mconfig 为适用于configs 集合的缩写。

本文定义的语义直接反映了Linux内核的make xconfig工具的行为, 在某些特定实例中，可能表现得与Kconfig语言开发人员最初所想要的不尽相同。比如为防止反向依赖，本文明确指出下面的声明: "应该小心地使用Select。Select 会强制一个符号为值而不考虑相关性。如果滥用 select，你可能会选择一个 foo 符号，即使 foo 依赖一个没设置的符号bar。"

Configs 是 Kconfig 模型中的第一个成分。一个 config 用类型、提示条件(定义 config 变得用户可变，一个默认值列表，一个表明自身反向依赖的表达式，这个条件会通过 select 语句强制启用这个特性)，和一个范围的集合(configs 或 hex 类型)定义了一个唯一的标识符类型。我们定义 Configs 如下:

Configs = Id × Type × KExpr(Id) × Default ∗ ×KExpr(Id) × P(Range) (3)

在这里,

• Type = {boolean, tristate, int, hex, string} 表示一个类型，就是config 的可能值。

• Default = KExpr(Id)×KExpr(Id) 表示默认值。 第一个 KExpr 表示一个默认表达式 (即赋值给符号的) 第二个 KExpr 表示默认值变得有效需要的条件。

• Range = (Int ∪ Hex ∪ Id) × (Int ∪ Hex ∪ Id) × KExpr(Id) 是上下界和条件的三倍。注意在范围的上下界少了 Tri; 这是因为如下段所述，范围在 int 和 hex 类型 configs 上才有效。

我们进一步定义一个函数 Id(m) 来表示在 model m 中的标识符:

Id(m) = {n | (n, \_, \_, \_, \_, \_) ∈ mconfig} (4)

Kconfig 第二个成分是 choice 结点的集合。一个 choice 是一个在配置中没定义符号的抽象概念，然而，它在其嵌套的元素上施加额外的约束。我们定义 choices 为一个类型的四倍组成，boolean 或者 tristate 是仅有的的有效类型，标记表明这个 choice 是强制的，随着一个表明其成员的标识符之后的提示条件。集合 Choices 定义为:

Choices = {boolean, tristate} × Bool × KExpr(Id) × P(Id(m)) (5)

**结构完整性规则**. 给定一个元素 (\_, t, \_, \_, rev, rngs) ∈ Configs, config 如果满足下列条件即是结构完整的:

• int、 hex 或者 string 类型 configs 的反向依赖必为 0t。换句话说，没有一个非 boolean 类型或者 tristate 类型的 config 会被选择。

(rev != 0t) =⇒ t = boolean ∨ t = tristate (6)

• 可以用数值类型(int 类型或 hex 类型)的 configs 唯一确定范围。因此，一个 config 若是结构完整的必有下列约束:

(|rngs| > 0) =⇒ t = int ∨ t = hex (7)

**简述具体语法转化**. Menuconfigs 和 menus 是 Kconfig 语言中具体语法的第一类概念，然而这些概念都没有出现在抽象语法中。首先, menuconfigs 与 configs 语义同源，只在配置程序方面有区别;因此，我们在抽象语法中模拟 menuconfigs 为 configs。

Menus 不定义符号，因此 menus 不在配置中。然而，menus 可以在其嵌套元素上施加约束。我们可以在所有嵌套符号上对提示、默认和范围条件进行语法的重写. 语法重写细节将在下文提供。

**2.语义**

**2.1 语义域**

一个Kconfig 模型的配置是一个值 v 的赋值，值 v ∈ Const 到 config 元素。 因此，所有可能的配置集合定义为:

Confs = Id → ⌊Const⌋ (8)

如果 c ∈ Confs 且 x ∈ Id, 我们为了在配置 c 下参考标识符 x 的值而写作 c(x)，现在我们在配置的集合方面定义了一个 Kconfig 模型的语义。

因此 P(Confs) 就是我们的语义域。我们定义 [[·]]kconfig 为求一个 Kconfig 模型值，并返回有效配置集合的函数:

[[·]]kconfig : Kconfig → P(Confs) (9)

**2.2 全局函数**

我们从一些完全符合语义的函数的定义开始。首先，我们使用 bool，用 boolean 逻辑定义了一个tristate 值的解释 : Tri → Bool, Bool = {T, F }:

 bool(v) = (10)

此外，我们定义了一个函数 access : (Id ∪ Const) × Confs → Const 取回一个常量或符号的值。当一个标识符有 的值(将在方程式14中定义)， 那么 access 函数以字符串形式返回标识符自身:

access(iv, c) = (11)

接下来，我们定义函数 toStr : Const → String 表示常量值到一个字符串表示的转化。在 toStr下列的定义中，设 i ∈ Int, h ∈ Hex 且 s ∈ String:

toStr(0t) = “n” toStr(1t) = “m” toStr(2t) = “y”

toStr(i) = “” + i toStr(h) = “0x” + h toStr(s) = s (12)

+ o操作符是字符串连接符。

最后，函数 eval : KExpr(Id) → Tri 描述在Kconfig语言中一个 KExpr 的求值。我们用e1, e2递归地定义 eval， e1, e2 ∈ KExpr(Id) 且 iv, ivx, ivy ∈ Id ∪ Const:

eval(ivx = ivy, c) =

eval(ivx != ivy, c) = 2t − eval(ivx = ivy, c)

eval(not e1, c) = 2t − eval(e1, c)

eval(e1 and e2, c) = min(eval(e1, c), eval(e2, c)) (13)

eval(e1 or e2, c) = max(eval(e1, c), eval(e2, c))

eval(iv, c) =

**2.3 估值函数**

**Kconfig模型**. 我们开始定义 [[·]]kconfig. Given给定一个 Kconfig 模型 m ∈ Kconfig, 这个模型的语义就是是涵盖这个 model, configs 和 choices 的所有表示的交集。换句话说，一个Kconfig模型的有效配置的集合就是满足所有表示的配置。 [[·]]kconfig : Kconfig → Confs 定义为:

kconfig =

(14)

**类型**. 第一个表示约束属于 config 类型施加的约束。一个 config 类型对于各个域限定其有效值。[[·]]type : Configs → Confs 定义为:

 (15)

**上下边界**. 接下来，边界表示一个 config 模拟的上下边界。下边界由一个 config 的反向依赖求值决定。联系到反向依赖用具体的语法模拟 select 语句的行为。

上边界被一个 config 的提示条件定义。这个表示对类型为 int, hex, 或 string 的 configs 无效，因为我们的结构完整性规则，反向依赖决定下边界的是 0t， eval 函数当评估一个值不在 Tri 中时返回 0t。

[[·]]bounds : Configs → Confs 定义为:

[[(n, , pro, , rev, )]]bounds =

{c ∈ Confs | eval(c(n), c) ≥ Lower(c) ∧ (Upper(c) < Lower(c) ∨ eval(c(n), c) ≤ Upper)} (16)

where Lower(c) = eval(rev,c) and Upper(c) = eval(pro,c).

**Defaults**. Kconfig 支持为一个 config 设置一个默认表达式.这个默认表达式与决定 config 什么时候用户的可变提示条件相互作用。当提示条件满足时，用户可以设置一个值。然而，当它不满足时，默认值就是这个 config 的值。 [[·]]default : Configs → Confs 定义为:

[[(n, \_, \_, defs, rev, \_)]]default =

{c ∈ Confs | bool(eval(pro, c)) ∨ c(n) = max(eval(default(defs, c)), eval(rev, c))} (17)

default: P(Default) × Type × Confs → Const 是一个默认模仿检索的函数. 联系到 defs 是一个默认值的列表 (且已排序). 一个默认值的效果依赖于它定义的 config 的类型. 如果 config 是boolean 或 tristate 类型, 那么默认值就被评估为 Tri. 否则， 默认值必为 Const 或 Id 中的值. 设 Nil 为空列表并且 :: 为列表 cons 操作符. 设 tTri ∈ {boolean, tristate} 并且tEntry ∈ {int, hex, string}. The default 函数被递归地定义， 所以我们开始定义它的基础情况

default(Nil, tTri, c) = 0t

default(Nil, tEntry, c) = “ ” (18)

方程式18声明给定一个默认的空列表，如果类型是boolean，或者tristate, 或者是 (int, hex 或 string类型的)空串之一， 我们返回 0t。接下来，我们定义递归规则。 在下面的方程式中，我们将 list 分解为首尾两部分。首先，我们描述 boolean 和 tristate 类型的函数:

default( (e, cond) :: rest, tTri, c) = (eval default( (e, crest, t ) Tri, c) if otherwise bool(eval(cond, c)) (19)

对于剩余类型:

default( (e, cond) :: rest, tEntry, c) = (access( default( e, c rest, t ) Entry, c) if otherwise bool(eval(cond, c)) (20)

**范围**. 范围利用 int 或 hex 的 configs 值的上下界。 [[·]]range : Configs → Confs 定义为:

[[n, \_, \_, \_, \_, rngs)]]range = {c ∈ Confs | ∀(l, u, cond) ∈ rngs.

bool(eval(cond, c)) → c(n) ≥ access(l, c) ∧ c(n) ≤ access(u, c)} (21)

**选择**. 选择限定了可被选择的成员的数量 (即有一个值大于 0t). 选择表示, [[·]]choice : Choices →

Confs 定义为:

[[(boolOrT ri, isMand, prompt, mems)]]choice =

{c ∈ Confs | bool(eval(prompt, c)) → Xor ∧ BChoice ∧ Mandatory} (22)

Xor 定义有且只有一个成员可被设置为 2t 的条件:

Xor = ∃m1 ∈ mems. (m1 = 2t) → (∀m2 ∈ mems \ {m1}. m2 = 0t) (23)

如果是一个 boolean 选择，那么其成员唯一的有效值为 2t. 与 Xor 结合, 定义了一个 boolean 选择可能至多有一个成员的值不等于 0t 且必为 2t:

BChoice = (boolOrT ri = boolean) → ∃m ∈ mems. c(m) = 2t (24)

最后，若这个选择是强制的，则至少选择一个成员:

Mandatory = isMand → ∃m ∈ mems. c(m) > 0t (25)

**模块** ,一个专用的模块用于支持内核中的模块。Disabling 模块不接受 1t 声明 configs 并有效的将所有 tristate configs 转变为 boolean configs。 一个专用的

符号 m 用于在表达式中用具体的语法标识一个模块特性的依赖。有 m 依赖的 Configs 不能被选择(即必为 0t)

如果没有选择模块， 我们假定专用的 m 标识符在抽象语法中被扩展为模块了。

[[m]]module = {c ∈ Confs | c(modules) = n → ∀i ∈ Id. c(i) != 1t} (26)

|  |  |  |
| --- | --- | --- |
|  | type | interpretation in tristate logic |
| X | tristate | X = y or X = m |
| !X | tristate | X = n |
| X | boolean | X = y |
| !X | boolean | X = n |
| X | string | X = “…”(some non-empty string) |
| !X | string | X = “” |
| X | int | X = i(some integer, including zero) |
| !X | int | X = “” |
| X | hex | X = i(some hex) |
| !X | hex | X = “” |

Table 1: Interpretation of propositional variables

**Undeclared symbols**. 我们也定义 undeclared symbols 的行为。Kconfig 语言支持引用我们在约束中没声明的符号。这些未声明的符号在我们的语义中被赋值为特殊符号 ⊥ 。在 2.2 节中的 eval 函数中，这个符号的用处将显而易见。 The [[·]]undeclared : Kconfig → P(Confs) 表示定义为:

[[m]]undeclared = {c ∈ Confs | ∀x ∈ Id \ Id(m). c(x) = ⊥} (27)

**3 1-Var 命题语义**

命题语义的目标是弱化完整语义的约束。

**重写表达式规则**. rewrite 是一个在表达式实现重写规则的部分函数. 函数 rewrite: KExpr(Id) → KExpr(Id) 定义为:



(28)

我们进一步设函数 relax: KExpr(Id) → KExpr(Id) 将表达式转化为 CNF，且移除子句相当于等式检查。T函数用于降低隐式先行词(LHS)的约束。

**语义**. 在命题语义中，设命题配置的集合为:

Confs p = Id → Bool (29)

函数 default 定义为 Id × Default ∗ ×Confsp → Bool. 命题语义需要一个额外的参数提供标识符声明。



(30)

默认表示定义为:

[[(n, t, vis, pro, defs, rev, rngs)]]default = {c ∈ Confsp | eval(pro, c) ∨ default(n, defs, c)} (31)

反向依赖和可见性条件施加的模型约束定义为:

[[(n, t, vis, pro, defs, rev, rngs)]]bounds = {c ∈ Confsp | (eval(relax(rev), c) → c(n))∧(c(n) → eval(vis, c))}

(32)

由于我们抽象各个 config 的值，故我们忽略范围. 我们也假定模块 config 被启用， 因此在 tristate configs 中允许 1t。

[[(boolOrT ri, isMand, vis, mems)]]choice =

(c ∈ Confs | eval(vis, c) → choose(1, ids(mems)) ∧ isMand → m∈\_ mems m!) (33)