基于 Maude 的 C 语言结构体初始化器合规性验证工具

陈实
51255902014

日期：2022 年 12 月 18 日

摘　要

代码合规性检查致力于确保关键领域的软件代码符合具备一定约束的代码标准，以达到安全、可靠的效果，并尽力避免系统进入非预期的状态。近年来，涌现了一些使用自动化工具进行代码合规性检查的尝试。目前，代码合规性检查正在逐渐成为医疗、汽车、航空航天乃至军用设备等领域的软件系统开发过程中的一个不可或缺的环节。

本文为《代数形式化方法》的课程报告，介绍了一个以基于 Maude 实现的检查器为核心的合规性验证工具的设计、开发过程，及其应用案例。该工具能够针对一类常见的合规性检查需求，也即 C 语言结构体初始化器的合规性，进行自动化的检查，以判断作为输入的 C 语言代码是否能够符合标准。受限于时间，并且考虑到 C 语言结构体初始化器的复杂性，本工具仅考虑了其全部特性的一个子集。在应用案例方面，以开源的 C 语言合规性检查工具 CppCheck 作为参照，比较了本工具与其在合规性检查方式与能力上的差异。

本文初步探索了代数形式化方法在代码合规性检查上的应用方式。

关键词：形式化验证，Maude，代码合规性检查

1　背景介绍

本章节将介绍本文工作的相关背景，包括代码合规性检查、C 语言的结构体初始化与重写逻辑。

1.1　代码合规性检查

代码合规性检查是一种以发现代码缺陷为目的的软件测试方法，是一种静态的、白盒的测试方法。传统的代码合规性检查依赖于领域专家，通过人工阅读代码的方式，理解代码的行为，仔细推敲代码中涉及到的控制逻辑、执行过程、算法、参数、系统调用等方面的内容，并判断其合规性。而合规性的约束内容，大致可以分为两个存在交集的不同抽象层次的类别，一种是针对编程语言特性的约束，另一种则是针对系统行为的约束。在这两种之中约束之中，前者并不涉及到或者较少涉及到对软件系统行为的理解，因而实际上有着更广泛的约束性，这也是本文设计的工具所针对的对象。

近年来，针对与具体软件系统行为关联较弱的编程语言特性的约束，工业界也出现了许多相关的标准。例如，C 语言是一种被广泛应用于各个嵌入式领域的编程语言，在有着高执行效率与简易语法的同时，由于其设计之初就考虑允许开发者控制几乎全部底层细节，向开发者暴露了过多的复杂性，而灵活的语法表达无疑加重了这个问题，因而在实际应用中，开发者们往往难以始终保持良好的编码习惯，或是完全理解某些语言特性，进而导致软件缺陷频发。

针对这一问题，英国的汽车工业软件可靠性联会 (The Motor Industry Software Reliability Association, MISRA) 曾在 2013 年发布《MISRA C:2012》[1] 这一 C 语言开发标准，其包括 143 条针对 C 语言特性进行约束的规则，以及 16 条与开发流程或程序相关的指令。其中，每条规则或指令都被归类为“强制”“必

须”和“建议”这三者之一，根据规则的影响范围划分为了针对“编译单元”或是“系统”这两个类别，并按照可判别性分类为“可判别”与“不可判别”。举例来说，某些嵌入式领域的软件产品会声称其符合所有的《MISRA C:2012》规则与指令，并将其作为产品的卖点之一，比如 FreeRTOS 等。在我国，也有着诸如 GJB 5369-2005《航天型号 C 语言安全子集》与 GJB 8114-2013《C/C++ 语言编程安全子集》等以提高军用软件的安全性为目的的软件编程标准。这些标准都可以作为静态检查的依据，以便为以嵌入式系统为主的关键领域的软件质量提供安全性与质量的保障。

1.2 C 语言结构体初始化器

C 语言标准 [2] 规定结构体 (Struct) 必须以一个非空的、由大括号包围的、由逗号分割的、以初始化器 (Initializer) 作为成员的初始化器进行初始化。其中，基本数据类型的初始化器可以是基本数据类型的值，而初始化器自身也可以互相嵌套，形成一个树形结构，其中每一个非叶子结点都是一个初始化器。

在 C99 标准中，指定初始化器 (Designator) 这一概念被引入，由于其有着较复杂的规则，且受限于文章篇幅，本文不对其进行考虑。另外，为简化讨论与编程难度，在结构体所允许使用的数据类型中，本文仅考虑字符型 (char) 这一最常见且被广泛应用的数据类型，也即无符号八位整形 (uint_8)。由于一维的字符型数组存在着字符串字面量的表达形式，若能够对其进行合规性检查，实际上将覆盖大部分实现时容易出错的情况。其它数据类型与更高维度的数组的初始化器检查可以视作是本文处理对象的推广形式，可以较容易的实现。

本文所考虑的 C 语言结构体初始化器所应遵守的规范，可以认为是《MISRA C:2012》中针对编译单元的必须类可判别规则 Rule 9.2(The initializer for an aggregate or union shall be enclosed in braces) 的一个子集，或者也可视作是 GJB 5369-2005《航天型号 C 语言安全子集》的强制类细则 13.1.3(结构体变量初始化的嵌套结构必须与定义的相一致) 的一个子集。这两个规范的描述看似不尽相同，但实际上前者与后者之间存在着包含关系。容易证明，若能够确保结构体的初始化器都按规范被大括号包围，那么对于嵌套的结构体来说，其初始化器的嵌套结构必然与其定义的嵌套结构完全一致。

1.3 Maude 语言与重写逻辑

本文实现的合规性检查工具的核心是一个使用 Maude 语言 [3] 实现的检查器。Maude 语言是一种基于重写逻辑（或称为项重写系统）的编程语言。重写逻辑可以直观方便地表达迁移系统的状态与状态之间的变迁过程，进而可用于描述 lambda 演算、时序逻辑与组合逻辑等更复杂的系统。其已经被广泛应用于针对各类系统的形式化验证方法之中。

2 建模与工具设计

在本章节中，将首先介绍 Clang AST 也即 Clang 编译器的 C/C++ 语言编译前端所使用的抽象语法树结构标准，随后介绍在这一结构上的针对 C 语言结构体初始化器的合规性检查器的设计与实现，并阐述开发过程中所使用的测试手段与样例。

2.1 抽象语法树建模

譬如，针对如下包含了结构体声明与结构体初始化器的 C 语言代码文件 input.c：

```
struct st {
    char a[4]; char b ; char c[4];
} s = {
    "asd",  1, {2, 3, 4, 5}
```

```
};
```

Clang 编译器的 C/C++ 编译前端会首先将代码进行分词，并转化为如下格式的抽象语法树结构（省去了部分与本文无关的细节）：

```
TranslationUnitDecl
|-RecordDecl <input.c:1:1, line:3:1> line:1:8 struct st definition
| |-FieldDecl <line:2:5, col:13> col:10 a 'char [4]'
| |-FieldDecl <col:16, col:21> col:21 b 'char'
| `-FieldDecl <col:25, col:33> col:30 c 'char [4]'
`-VarDecl <line:1:1, line:5:1> line:3:3 s 'struct st':'struct st' cinit
  `-InitListExpr <col:7, line:5:1> 'struct st':'struct st'
    |-StringLiteral <line:4:5> 'char [4]' "asd"
    |-ImplicitCastExpr <col:13> 'char' <IntegralCast>
    | `-IntegerLiteral <col:13> 'int' 1
    `-InitListExpr <col:16, col:27> 'char [4]'
      |-ImplicitCastExpr <col:17> 'char' <IntegralCast>
      | `-IntegerLiteral <col:17> 'int' 2
      |-ImplicitCastExpr <col:20> 'char' <IntegralCast>
      | `-IntegerLiteral <col:20> 'int' 3
      |-ImplicitCastExpr <col:23> 'char' <IntegralCast>
      | `-IntegerLiteral <col:23> 'int' 4
      `-ImplicitCastExpr <col:26> 'char' <IntegralCast>
        `-IntegerLiteral <col:26> 'int' 5
```

观察该抽象语法树即可以发现，这一条语句从最顶层来看包含两个声明结点，分别为 RecordDecl 结点与 VarDecl 结点，其中 RecordDecl 结点是由 FieldDecl 结点组成的列表（实际上，它是也可以包含 RecordDecl 结点的广义列表）；而如若忽略其中的类型转换结点，那么 VarDecl 结点也可以看作是由 StringLiteral 结点、IntegerLiteral 结点与 InitListExpr 结点所组成的广义列表。其中，InitListExpr 结点实际上也是由 IntegerLiteral 结点组成的列表。

因此，可以分别定义 il、sl 为以自然数、字符串为输入，IntegerLiteral 结点、StringLiteral 结点为输出的构造函数，如此便可以由 Maude 内建的自然数 (Nat) 与字符串 (String) 类型方便的构造出符合 Clang AST 结构的 RecordDecl 结点与 VarDecl 结点，以便为合规性检查器提供基础。具体来说，可定义如下模块 C-AST：

```
fmod C-AST is
    --- Type names follow Clang AST
    protecting BOOL .
    protecting NAT .
    protecting STRING .

    sort FieldDecl FieldDeclList .
    subsort FieldDecl < FieldDeclList .
    op char_; : String -> FieldDecl [ctor] .
    op char_[_]; : String Nat -> FieldDecl [ctor] .
    op __ : FieldDeclList FieldDeclList -> FieldDeclList [ctor assoc] .

    sort RecordDecl . --- As the root node of the struct definition
    subsort RecordDecl < FieldDecl .
    op struct{_}_; : FieldDeclList String -> RecordDecl .

    sorts InitList InitListExpr .
    sort IntegerLiteral StringLiteral .
```

```
    subsort IntegerLiteral < InitList .
    subsort StringLiteral < InitList .
    op il(_) : Nat -> IntegerLiteral .
    op sl(_) : String -> StringLiteral .

    --- Build InitListExpr as a tree-like structure
    subsort InitListExpr < InitList .
    op _,_ : InitList InitList -> InitList [ctor assoc] .
    op {_} : InitList -> InitListExpr [ctor] .

    sort VarDecl . --- As the root node of the variable initialization
    subsort InitListExpr < VarDecl .
endfm
```

2.2 合规性检查器建模

前文提到，本文设计的合规性检查器需要检查的规则是，结构体始化器的嵌套结构必须与其定义的嵌套结构完全一致。该规则看似非常平凡，但实际上由于 C 语言表达的灵活性，需要正确处理各种边界情况和一些例外情况。如果使用传统的命令式编程语言，在实现时会非常容易出错（后文的案例会证实这一点）。

首先，考虑所有可能的情况。RecordDecl 需要与 VarDecl 相对应。而 FieldDecl 又分为单变量与数组两种类型，前者与 IntegerLiteral 相对应，后者则相对灵活，可以与 StringLiteral 或是 InitListExpr 相对应。

那么，可以定义以 FieldDecl 和 IntegerLiteral/StringLiteral/InitListExpr 这三者之一作为输入的 checkLeaf 函数。该函数负责检查那些不包含嵌套结构体的结构体成员与其对应的初始化器是否合规。对于单变量型 FieldDecl，其只能接受 IntegerLiteral，而对于数组型 FieldDecl，其能够接受 StringLiteral/InitListExpr 这两种类型。对于前者，可直接通过 length 函数进行判断；而对于后者，则可通过不断同时减少数组长度与 InitListExpr 中 IntegerLiteral 个数的方式，尝试将其规约至前者数组长度为一，且后者只包含一个 IntegerLiteral 的情况来进行检查。此外，要注意在合规情况下的字符串长度应等于字符数组的长度减去一（由结尾的字符'\0' 产生）。

接着，考虑输入分别为 RecordDecl 和 VarDecl 的情况，也即检查器的核心功能 check 函数。可以很自然的考虑分治的处理方式：在第一次检查中，从 RecordDecl 中拿出第一个元素，判断其类型，如果是 RecordDecl，那么将其与 VarDecl 中的第一个元素一同作为参数递归调用 check 函数自身，否则调用上文提到的 checkLeaf 函数。在第二次检查中，再调用自身检查排除了各自列表中第一个元素的新的 RecordDecl 和 VarDecl。如果遇到一些类型无法匹配的情况，则返回 false，反之，则返回这两次检查结果的逻辑与运算的结果。若 RecordDecl 与 VarDecl 均为空，则返回 true。那么，通过这种方式，就可以检查结构体始化器是否合规。

该方法的正确性是显然的，受限于篇幅，本文不给出形式化的证明方式。具体来说，可根据该方法定义如下模块 C-INITIALIZATION-CHECKER：

```
fmod C-INITIALIZATION-CHECKER is
    protecting C-AST .
    protecting BOOL .
    protecting NAT .
    protecting STRING .

    op checkLeaf(_==_) : FieldDecl IntegerLiteral -> Bool .
    op checkLeaf(_==_) : FieldDecl StringLiteral -> Bool .
    op checkLeaf(_==_) : FieldDecl InitListExpr -> Bool .
```

```
    var S S' : String .
    var N : Nat .
    var IL IL' : IntegerLiteral .
    var SL : StringLiteral .
    var ILS : InitList .
    var ILE : InitListExpr .

    eq checkLeaf( char S ; == IL ) = true .
    eq checkLeaf( char S ; == sl(S') ) = length(S') == 0 . --- This char is '\0'
    eq checkLeaf( char S ; == { ILS } ) = false .
    eq checkLeaf( char S[N] ; == IL ) = false .
    eq checkLeaf( char S[N] ; == sl(S') ) = ( N == length(S') + 1 ) .
    eq checkLeaf( char S[s N] ; == { IL, ILS } ) = checkLeaf(char S[N] ; == { ILS }) .
    eq checkLeaf( char S[1] ; == { IL } ) = true .
    eq checkLeaf( char S[2] ; == { IL } ) = false .
    eq checkLeaf( char S[0] ; == { IL } ) = false .

    op check(_==_) : RecordDecl VarDecl -> Bool .

    var FD FD' : FieldDecl .
    var FDL : FieldDeclList .

    --- Boundaries
    eq check( struct { FD } S ; == { IL, ILS } )  = false .
    eq check( struct { FD } S ; == { SL, ILS } )  = false .
    eq check( struct { FD } S ; == { ILE, ILS } )  = false .

    eq check( struct { FD FD' } S ; == { IL } )  = false .
    eq check( struct { FD FD' } S ; == { SL } )  = false .
    eq check( struct { FD FD' } S ; == { ILE } )  = false .

    eq check( struct { FD } S ; == { IL } ) = checkLeaf(FD == IL) .
    eq check( struct { FD } S ; == { SL } ) = checkLeaf(FD == SL) .
    eq check( struct { FD } S ; == { ILE } ) = check(FD == ILE) .

    --- Reduce rules
    eq check( struct { FD FDL } S ; == { IL, ILS } ) =
        checkLeaf( FD == IL ) and check( struct { FDL } S ; == { ILS } ) .
    eq check( struct { FD FDL } S ; == { SL, ILS } ) =
        checkLeaf( FD == SL ) and check( struct { FDL } S ; == { ILS } ) .
    eq check( struct { FD FDL } S ; == { ILE, ILS } ) =
        check( FD == ILE ) and check( struct { FDL } S ; == { ILS } ) .
    eq check(char S [N]; == { ILS }) = checkLeaf(char S [N]; == { ILS } ) .
    eq check(char S ; == { ILS }) = false .
endfm
```

2.3 合规性验证工具设计

尽管该合规性检查器的结构与基本原理足够清晰直观，在开发期间，难免会存在由开发人员的失误而引入的错误。为了能够寻找出这些错误，进而保证检查器本身的正确性，本文设计了一个可用于自动化回归测试的简单 Python 脚本。该工具脚本使用了 Maude 的 Python 程序接口，通过不断从测试样例文件中读取推导内容，将其输入上文实现的合规性检查工具进行测试，并比较实际结果与期望的推导结果

是否一致，来测试开发过程中可能存在的由失误引入的错误。具体来说，该工具的实现如下：

```
import maude, os

maude.init()
for root, subdirs, files in os.walk("tests"):
    for filename in files:
        if not filename.endswith(".test"):
            continue
        file_path = os.path.join(root, filename)
        with open(file_path, 'r') as f:
            num = 0
            f_content = f.readlines()
            for i in range(len(f_content)):
                line = f_content[i].strip()
                if line.startswith("load:"):
                    maude.load(line.replace("load:", ""))
                if line.startswith("module:"):
                    m = maude.getModule(line.replace("module:", ""))
                if line.startswith("reduce:"):
                    t = m.parseTerm(line.replace("reduce:", ""))
                    t.reduce()
                    r = str(t)
                if line.startswith("result:"):
                    expected = line.replace("result:", "")
                    assert expected == r, f"In {file_path} line {i+1}: Expected: {expected},
                        Actual: {r}"
                    num += 1

        print(f"Pass {num} tests in:", file_path)
```

该回归测试工具能够读取如下形式的测试脚本，并对其中的 reduce 语句进行逐条执行（其它测试用例详见代码库）。

```
load:src/c-init-checker.maude
module:C-INITIALIZATION-CHECKER

reduce:check(struct {char "a"[2] ; struct {char "a"[2] ; char "a"[2] ; } "s" ; } "s" ; == {{il
    (1),il(2)},{{il(1),il(2)},{il(1),il(2)}}})
result:true

reduce:check(struct {struct {char "a"[2] ; char "a"[2] ; } "s" ; char "a"[2] ; } "s" ; == {{{
    il(1),il(2)},{il(1),il(2)}},{il(1),il(2)}})
result:true

reduce:check(struct {struct {char "a"[2] ; struct {char "a"[2] ; char "a"[2] ; } "s" ; } "s" ;
     char "a"[2] ; char "a" ; } "s" ; == {{{il(1),il(2)},{{il(1),il(2)},{il(1),il(2)}}},{il(1)
    ,il(2)},il(1)})
result:true

reduce:check(struct {struct {char "a"[2] ; struct {char "a"[2] ; char "a"[2] ; } "s" ; } "s" ;
     char "a"[2] ; } "s" ; == {{{il(1),il(2)},{{il(1),il(2)},{il(1),il(2)}},{il(1),il(2)}}})
result:false
```

受限于时间与篇幅，笔者暂未实现直接从 C 语言代码生成合规性检查器输入的自动转化工具。尽管

该转换过程很容易手工实现，在实际处理时，仍需要对 C 语言代码进行预处理与分词处理。未来将考虑实现这一工具，从而打通直接针对 C 语言代码进行检查的流程。

3 应用案例

前文提到，针对“结构体始化器的嵌套结构必须与其定义的嵌套结构完全一致”这一规则，由于 C 语言表达的灵活性，如果使用命令式编程语言，在实现时会非常容易出错。这里给出一个具体的例子。

Cppcheck 是一个著名的开源 C/C++ 代码缺陷静态检查工具。它能够针对包括《MISRA C:2012》在内的诸多编码标准中的大部分可判别规则进行静态检查，当然也包括前文提到的本文设计的合规性检查对象的推广形式也即《MISRA C:2012》的 Rule 9.2。令人遗憾的是，在测试和比较中，笔者发现，虽然 Cppcheck 能针对更多类型或是内容进行合规性检查，并有着规模不小的回归测试用例，但即使是在其最新公开发布于 2022 年 8 月 28 日的 Cppcheck-2.9 版本中，若使用前文提到的 input.c 这一看上去无比简单的源程序代码，即可让其检查过程直接崩溃。

图 1: Cppcheck 在面对前文提到的 input.c 时输出的崩溃提示

```
Traceback (most recent call last):
  File "addons/misra.py", line 4735, in <module>
    main()
  File "addons/misra.py", line 4677, in main
    checker.parseDump(item)
  File "addons/misra.py", line 4333, in parseDump
    self.executeCheck(902, self.misra_9_2, cfg)
  File "addons/misra.py", line 4244, in executeCheck
    check_function(*args)
  File "addons/misra.py", line 2104, in misra_9_2
    misra_9.misra_9_x(self, data, 902)
  File "/home/kale/cppcheck/addons/misra_9.py", line 424, in misra_9_x
    parser.parseInitializer(ed, eq.astOperand2)
  File "/home/kale/cppcheck/addons/misra_9.py", line 323, in parseInitializer
    if child.elementType != 'record' or self.token.valueType.type != 'record' or child.valueType.typeScope != self.token.valueType.typeScope:
AttributeError: 'NoneType' object has no attribute 'elementType'
```

经过长时间的调试，笔者发现其崩溃的根本原因是，尽管针对这一规则，Cppcheck 也使用了类似的依次检查的方法，但其错误地处理了 VarDecl 中包含 StringLiteral 时的情况，使得样例程序中的 a[1] 与数值 2 进行了匹配，进而导致后续检查过程无法继续进行。这一结果也从侧面说明了使用 Maude 进行形式化建模编写的检查器相对于采用传统的命令式编程语言编写的检查器存在一定的优越性。

当然，如果后续时间允许，笔者也会考虑向 Cppcheck 提交这一错误的触发方式、修复方式与回归测试用例。

4 总结

本文介绍了一个以基于 Maude 实现的检查器为核心的合规性验证工具的设计、开发过程，及其应用案例。该工具能够针对 C 语言结构体初始化器的合规性的子集进行自动化检查。随后，比较了该工具与开源的代码静态分析工具 Cppcheck 之间的差异，展现了以 Maude 为实现手段的代数形式化建模方法在代码合规性检查领域所具有的应用潜力。

参考文献

[1] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013. ISBN: 9781906400101. URL: https://books.google.com.hk/books?id=3yZKmwEACAAJ.

[2] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[3] Francisco Dur´an Manuel Clavel. *Maude Manual (Version 3.2.1)*. SRI International, Menlo Park, CA 94025, USA: Springer, 2022.