一种基于安全C的编译器形式验证方法

**陈志伟 马殿富 谭宇**

（北京航空航天大学计算机学院 北京 100191）

**摘 要** 编译器是重要的系统开发工具，其安全可靠性对安全攸关软件的开发有着重要影响。传统检测编译错误的方法是进行大量的测试，但测试存在局限性且难以达到完全覆盖。近年来，形式化验证方法在编译器的验证中得到了广泛的关注，但当前的形式验证方法却存在证明复杂度高、验证能力弱、算法效率不够高等问题。本文提出了一种基于安全C子集的编译器形式验证方法，通过从安全C子集中抽象出C文法单元的概念，把对编译器的形式验证转化为了对有限的C文法单元的验证。文中引入了一阶逻辑的公理系统和专用公理，在此公理系统上通过定理证明的方式，完成了C文法单元和目标码模式的语义一致性验证，从而完成了对编译器的形式验证的过程。

**关键词** 形式验证 C文法单元 目标码模式 指称语义 语义一致性

# 0 引言

随着计算机应用技术的飞速发展，软件在国民经济和国防建设的各个领域中得到了广泛的应用。安全攸关软件，如航空机载软件，作为各类安全关键系统的重要构成部分，其内部结构越来越复杂、应用环境越来越开放，这些因素使得人们更加关注其安全可靠性。因此，对安全攸关软件尤其是大型客机机载软件进行安全性分析、设计以及验证变得尤为重要。

目前航空领域中主要采用的验证标准是美国航空无线电委员会（RTCA）于1992年12月发布的航空适航认证标准体系DO-178B[1]《机载系统和设备认证中的软件要求》标准。DO-178B规定了机载软件的设计和开发进程，并描述目标的可追踪性过程，按照可能引起航空器不同的失效状态将机载软件划分为A、B、C、D、E五个软件等级，分别对应灾难性的、严重的、较重的、较轻的和无影响的五类失效状态。RTCA于2012年又发布了DO-178C[2]。DO-178C对DO-178B的补充有四个方面：软件工具验证、基于模型的开发和验证、面向对象编程、形式化方法。在软件开发新技术、新方法不断涌现的今天，这些补充和修订很好的适应了当前安全攸关软件的开发过程。

编译器作为软件开发过程中的重要工具，是实现软件从设计到在硬件上运行的桥梁，如何确保编译器编译过程的正确性是进行软件开发所面临的重要难题。传统检测编译错误的方法是进行大量的测试，但是测试只能证明软件是有错的，不能证明软件是没有错误的。近年来，形式化验证方法在编译器的验证中得到了持续的关注。形式化验证方法基于严格的数学理论，将软件系统和性质都用逻辑方法来规约，通过基于公理和推理规则组成的形式系统，以如同数学中定理证明的方法来证明软件系统是否具备所期望的关键性质。当前对编译器进行形式验证的方法主流的有两种：一种是模型检测法[3]，它的主要瓶颈是状态爆炸问题，所以仅适用于对有限的、状态空间较小的系统进行分析；另一种是定理证明技术[4]，目前比较好的证明方式是使用编程和证明统一的框架，如PVS、Coq和Isabelle等，其局限性在于要求工具使用者具备较高的理论基础和技术背景，限制了大多数程序设计人员的学习和使用。

本文提出了一种基于安全C子集的编译器形式验证方法。通过引入MISRA-C[5]，即汽车制造业嵌入式C编码标准，对安全攸关系统中使用的C语言进行限制，由该规范定义的Ｃ语言被认为是易读、可靠、可移植、易于维护的。将MISRA-C与航天型号软件的特点相结合，重新定义了一系列C语言软件的编程准则，形成了C安全子集。C安全子集严格要求了编译器的成熟度及稳定性，编译器必须忠实地反映源语言的代码结构和语义，以方便编译前后的代码审查、比较和追踪。从安全C子集中，可以得到有限的C文法单元，通过文法单元对应的下推自动机可以识别出源程序，一个正确的源程序必须符合文法单元的规定。因此，对于源程序编译过程正确性的证明，就可以转化为文法单元语义的等价性证明，这种方法大大简化了传统形式验证方法中存在证明复杂度高和耗费的时间长等问题。

# 1 相关工作

对程序的定理证明研究开始于20世纪60年代Hoare[6]和Floyd[7]发表的论文。Hoare在他的论文里提出一个形式系统，称作霍尔逻辑。在霍尔逻辑中存在一组证明规则，称为霍尔规则。用霍尔规则进行推导能得到部分正确性断言的形式化证明，所以霍尔逻辑能用于机器证明。

分离逻辑是对霍尔逻辑的一个扩展，通过提供表达显示分离的逻辑连接词以及相应的推导规则，消除了共享的可能，能够以自然的方式来描述计算过程中内存的属性和相关操作, 从而简化了对指针程序的验证工作。分离逻辑被证明具有更强的验证能力，使得程序验证和推理技术前进了一大步。

近年来，比较具有代表性的是Leroy X[8~9]领导的CompCert项目组所做的工作，他们首次完成了对一个完整且实际的编译过程的正确性形式验证，整个证明过程完全形式化的且是机器自动生成的。为了支持自动的形式化验证，CompCert采用了辅助定理证明工具Coq Assistant对编译过程进行重新构造。整个过程由八种不同的中间语言之间的转换构成，然后借助Coq Assistant对整个编译过程的正确性即语义可保持性进行证明。目前CompCert编译器只能实现对一个C语言子集的编译验证，还不能完全覆盖所有C语言元素，且后端优化程度还比较低，项目也正在进一步研究中。

2011年，Yang X[10]等人在关于Csmith，一个自动测试用例生成工具，的研究工作中对主流的C编译器进行测试，共向编译器的开发者报告了325个原来未知的bugs，其中包括著名的Intel CC，GCC和LLVM编译器等。在所有被比较的11种开源或商用的C编译器中，CompCert表现非常突出，在6个CPU年中，没有找到任何wrong-code错误。

# 2 形式验证方法

## 2.1 C文法单元和语义

抽象语法树（abstract syntax code，AST）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种语法结构。源程序与其对应的抽象语法树本质上是等价的，抽象语法树中包含了源程序的所有语义信息，于是对源程序的验证可以转换为对抽象语法树的验证。又由树的定义可知：每棵树都是由其子树递归定义的，所以可以通过分别对语法子树的验证来完成对整个语法树的验证。

在安全C子集的约束下，去除掉C语言中不安全的语法规范，如指针等。对抽象语法树进行抽象和归纳，可以得到一些公共的子语法树结构，把这些子语法树结构提取出来，并还原成C语言的形式，就得到了C文法单元。对源程序的编译验证等价为对每个C文法单元的验证，可以通过验证编译前后每个C文法单元和对应的目标代码模式的语义是否保持一致来实现。

为了获得每个C文法单元的语义，本文引入了语境的概念，根据语境可以定义出的文法单元的语义。语境表示待证明序列中每一个证明项所在的环境和上下文，其中包括函数局部变量、 全局变量以及上下文等。下表1给出了部分C文法单元和其对应的语义。

表1 C文法单元和语义

|  |  |  |
| --- | --- | --- |
| 语句 | C文法单元 | C文法单元语义 |
| <if-statement> | if (<LOG-EXP>)  {  <STA-LIST\_1>  }  else  {  <STA-LIST\_2>  } | σ(<LOG-EXP>) -> σ(<STA-LIST\_1>)  ~σ(<LOG-EXP>) -> σ(<STA-LIST\_2>) |
| <while-statement> | while (<LOG-EXP>)  {  <STA-LIST>  } | {σ(<LOG-EXP>) -> σ(<STA-LIST>)} \*\* n  ~σ(<LOG-EXP>) -> skip |
| <do-while-statement> | do  {  <STA-LIST>  } while (<LOG-EXP>); | σ(<STA-LIST>)  {σ(<LOG-EXP>) -> σ(<STA-LIST>)} \*\* n  ~σ(<LOG-EXP>) -> skip |
| <for-statement> | for(<ASS-EXP\_1>;  <LOG-EXP>;  <ASS-EXP\_2>)  {  <STA-LIST>  } | σ(<ASS-EXP\_1>)  {σ(<LOG-EXP>) -> σ(<STA-LIST>);  σ(<ASS-EXP\_1>)} \*\* n  ~σ(<LOG-EXP>) -> skip |

表中，σ符号代表着取值的过程，σ(<LOG-EXP>)表示获得逻辑表达式的值，按照安全C子集规范，逻辑表达式的值只能为0和1。<STA-LIST>表示语句块，可以包括表达式语句、条件选择语句等，一般把其交给识别语句块的下推自动机进行递归处理。<ASS-EXP>表示赋值语句，其取值后的返回值就为表达式的值。“{..} \*\* n” 代表着循环执行大括号内的语句，用来定义循环语句的语义。skip表示直接跳转到下一条语句进行执行，在32位的Power PC指令集下，定义skip等于σ(PC = PC + 4)，PC表示程序计数器。

## 2.2 指称语义

指称语义是采用形式系统方法，用相应的数学对象（如set, function等）对一个即定形式语言的语义进行注释的学问。指称语义还可以解释为：存在着两个域，一个是语法域，在语法域中定义了一个形式语言系统；另外一个是数学域（或称之为已知语义的形式系统）。在32位的Power PC指令集范围内，本文根据Power PC官方文档[11]中给出的每条指令的操作语义，为汇编指令建模并得到对应的指称语义。下表2给出了部分Power PC汇编指令的指称语义。

表2 指称语义

|  |  |  |
| --- | --- | --- |
| 指令 | 指令用法 | 指称语义 |
| li | li rD,SIMM | GPR[rD] = SIMM |
| lwz | lwz rD,D(rA) | GPR[rD] = MEM[D] |
| stw | stw rS,D(rA) | MEM[D] = GPR[rS] |
| b | b target | PC = PC + @target |
| beq | beq crfD,target | CR[crfD] == b100 -> PC = PC + 4  CR[crfD] == b010 -> PC = PC + 4  CR[crfD] == b001 -> PC = PC + @target |
| bne | bne crfD,target | CR[crfD] == b100 -> PC = PC + @target  CR[crfD] == b010 -> PC = PC + @target  CR[crfD] == b001 -> PC = PC + 4 |
| cmp | cmp crfD,L,rA,rB | GPR[0] < 0 -> CR[7] = b100  GPR[0] > 0 -> CR[7] = b010  GPR[0] == 0 -> CR[7] = b001 |
| cmpi | cmpi crfD,L,rA,SIMM | GPR[rA] < SIMM -> CR[crfD] = b100  GPR[rA] > SIMM -> CR[crfD] = b010  GPR[rA] == SIMM -> CR[crfD] = b001 |
| add | add rD,rA,rB | GPR[rD] = GPR[rA] + GPR[rB] |
| addic | addic | GPR[rD] = GPR[rA] + SIMM |
| subf | subf rD,rA,rB | GPR[rD] = - GPR[rA] + GPR[rB] |
| mullw | mullw rD,rA,rB | GPR[rD] = GPR[rA] \* GPR[rB] |
| divw | divw rD,rA,rB | GPR[rD] = GPR[rA] / GPR[rB] |

表中，GPR（General-Purpose Register）表示Power PC的通用寄存器，主要用作堆栈指针、函数的第一个参数和返回值等。CR（Conditional Register）为条件寄存器，可以反映某些操作的结果（比如cmp指令），协助测试和分支转移指令的执行。MEM为内存空间，存储了局部、全局等变量的值。@target代表相对地址，一般用在跳转指令中，PC = PC + @target表示从当前PC所指向的地址跳转到target标识的地址，PC = PC + 4表示直接执行下一条指令，在32位Power PC指令集，32位正好为4个字节。

## 2.3 目标码模式和命题

目标码模式是通过GCC编译器编译在一定语境下的C文法单元得到目标码序列，消除掉语境对目标码序列的影响而获得的目标码序列的一般化（Generalize）表示形式。对编译器的形式化验证，最终需要转化为C文法单元的语义和目标码模式的语义一致性验证。下表3给出了在32位Power PC指令集下部分C文法单元对应的目标码模式。

表3 目标码模式

|  |  |  |
| --- | --- | --- |
| 语句 | C文法单元 | 目标码模式 |
| <if-statement> | if (<LOG-EXP>)  {  <STA-LIST\_1>  }  else  {  <STA-LIST\_2>  } | <LOG-EXP>  cmpi 7,0,0,0  beq 7,.L1  <STA-LIST\_1>  b .L2  .L1:  <STA-LIST\_2>  .L2: |
| <while-statement> | while (<LOG-EXP>)  {  <STA-LIST>  } | b .L2  .L1:  <STA-LIST>  .L2:  <LOG-EXP>  cmpi 7,0,0,0  bne 7,.L1 |
| <do-while-statement> | do  {  <STA-LIST>  } while (<LOG-EXP>); | .L1:  <STA-LIST>  <LOG-EXP>  cmpi 7,0,0,0  bne 7,.L1 |
| <for-statement> | for(<ASS-EXP\_1>;  <LOG-EXP>;  <ASS-EXP\_2>)  {  <STA-LIST>  } | <ASS-EXP\_1>  b .L2  .L1:  <STA-LIST>  <ASS-EXP\_2>  .L2:  <LOG-EXP>  cmpi 7,0,0,0  bne 7,.L1 |

程序的形式化证明需要特定的公理系统作为基础。公理系统（axiomatic system）就是把一个科学理论公理化，用公理方法研究它，每一科学理论都是由一系列的概念和命题组成的体系。基于2.2中Power PC汇编指令的指称语义，使用3.1中命题映射算法，可以得到每个目标码模式的命题。表4中给出了条件选择语句（<if-statement>）和循环语句（<while-statement>）目标码模式命题。

表4 目标码模式命题

|  |  |  |
| --- | --- | --- |
| 语句 | 目标码模式 | 目标码模式命题 |
| <if-statement> | <LOG-EXP>  cmpi 7,0,0,0  beq 7,.L1  <STA-LIST\_1>  b .L2  .L1:  <STA-LIST\_2>  .L2: | P1: GPR[0] = <LOG-EXP>  P2: (GPR[0] < 0 -> CR[7] = b100) || (GPR[0] > 0 -> CR[7] = b010) || (GPR[0] == 0 -> CR[7] = b001)  P3: (CR[7] == b100 -> PC = PC + 4) || (CR[7] == b010 -> PC = PC + 4) || (CR[7] == b001 -> PC = PC + @.L1)  P4= <STA-LIST\_1>  P5: PC = PC + @.L2  P6: .L1:  P7: <STA-LIST\_2>  P8: .L2: |
| <while-statement> | b .L2  .L1:  <STA-LIST>  .L2:  <LOG-EXP>  cmpi 7,0,0,0  bne 7,.L1 | P1: PC = PC + @.L2  P2: .L1:  P3: <STA-LIST>  P4: .L2:  P5: GPR[0] = <LOG-EXP>  P6: (GPR[0] < 0 -> CR[7] = b100) || (GPR[0] > 0 -> CR[7] = b010) || (GPR[0] == 0 -> CR[7] = b001)  P7: (CR[7] == b100 -> PC = PC + @.L1) || (CR[7] == b010 -> PC = PC + @.L1) || (CR[7] == b001 -> PC = PC + 4) |

## 2.4 推理证明

本文提出的形式化验证方法是基于一阶逻辑的公理系统，从公理系统中事先给定的公理（如，目标码模式命题）出发，根据推理规则推导出一系列新命题，并作为前提加入到之后的证明过程中。由于证明序列中的每一项都是前提、公理或者定理，又因为一阶逻辑的公理系统是可靠的，所以证明序列中的每一项一定是正确的，从而最终推导出来证明序列一定是正确的。

MP(Modus Ponens)规则，被称为分离论证或分离规则（Rule of Detachment）是一阶逻辑的公理系统中的最基本的推理规则。MP 规则可表示为：

p → q, p ⊢ q

其含义是: 如果 p成立, 那么 q成立, 又p成立, 因此q成立。分离规则由三个陈述（或命题，或语句）组成：第一个陈述是一个条件陈述，即p蕴涵q；第二个陈述是p，即条件陈述的前提为真。从前两个陈述就能逻辑上推出q，即条件陈述的结论也必定为真。

CI(Conjunction Introduction)规则，被称为合取引入或组合规则。CI规则可表示为：

p, q ⊢ p∧q

其含义是：若p, q成立，则p∧q成立。合取规则主要用来把多个为真的命题转化为单一的命题。

在实际的验证过程中，对于表达式文法单元、条件选择文法单元等的目标码模式命题，由于它们不含有循环结构，运用MP规则和一阶逻辑的公理系统中的公理集、定理集等，可以很方便的完成命题的推理证明。但是，对于循环结构，如<while-statement>的目标码模式命题进行直接推理后，得到的目标码模式的语义与C文法单元的语义差异较大，无法直接证明两者的语义是一致的，所以本文引入了限定数学归纳法对循环结构目标码模式命题进行证明。

限定数学归纳法的逻辑基础是自然数公理，也称皮亚诺公理[12]。一般数学归纳法的逻辑表达式为P(0)⋀(∀n)(P(n)→P(s(n))→(∀n)P(n)，限定数学归纳法是在一般数学归纳法的基础上，限定n是有穷的，即对于循环结构程序，循环是可终止的，终止条件由人给出。表5和表6将分别给出<if-statement>的目标码模式命题和<while-statement>的目标码模式命题的证明。

表5 <if-statement>的证明

|  |  |
| --- | --- |
| 证明 | 理由 |
| S1= GPR[0] = <LOG-EXP> | P1 |
| S2= (GPR[0] < 0 -> CR[7] = b100) || (GPR[0] > 0 -> CR[7] = b010) || (GPR[0] == 0 -> CR[7] = b001) | P2 |
| S3= (<LOG-EXP> < 0 -> CR[7] = b100) || (<LOG-EXP> > 0 -> CR[7] = b010) || (<LOG-EXP> == 0 -> CR[7] = b001) | S1, S2, MP |
| S4= (CR[7] == b100 -> PC = PC + 4) || (CR[7] == b010 -> PC = PC + 4) || (CR[7] == b001 -> PC = PC + @.L1) | P3 |
| S5= (<LOG-EXP> < 0 -> PC = PC + 4) || (<LOG-EXP> > 0 -> PC = PC + 4) || (<LOG-EXP> == 0 -> PC = PC + @.L1) | S3, S4, MP |
| S6= <STA-LIST\_1> | P4 |
| S7= PC = PC + @.L2 | P5 |
| S8= .L1: | P6 |
| S9= <STA-LIST\_2> | P7 |
| S10= .L2: | P8 |
| S11= {  (<LOG-EXP> < 0 -> PC = PC + 4) || (<LOG-EXP> > 0 -> PC = PC + 4) || (<LOG-EXP> == 0 -> PC = PC + @.L1) ∧  <STA-LIST\_1> ∧  PC = PC + @.L2 ∧  .L1: ∧  <STA-LIST\_2> ∧  .L2:  } | S5, S6, S7, S8, S9, S10, CI |
| S12= {  (<LOG-EXP> < 0 -> <STA-LIST\_1>) || (<LOG-EXP> > 0 -> <STA-LIST\_1>) || (<LOG-EXP> == 0 -> <STA-LIST\_2>)  } | S11 |

表5中，最终推导出的证明序列为S12，对S12进行取值（σ）操作得到的目标码模式的语义为：

σ(<LOG-EXP>) -> σ(<STA-LIST\_1>) || ~σ(<LOG-EXP>) -> σ(<STA-LIST\_2>)，

结合表1中<if-statement>的语义可知，二者语义保持了一致性，证毕。

对于<while-statement>证明序列的推理同<if-statement>，在此直接给出最终推导出的证明序列，并结合文法单元的语义对证明序列使用限定数学归纳法来证明。

表6 <while-statement>的证明

|  |
| --- |
| 命题：{σ(<LOG-EXP>) -> σ(<STA-LIST>)} \*\* n || ~σ(<LOG-EXP>) -> skip  引理：PC = PC + @.L2 ∧  .L1: ∧  <STA-LIST> ∧  .L2: ∧  (<LOG-EXP> < 0 -> PC = PC + @.L1) || (<LOG-EXP> > 0 -> PC = PC + @.L1) || (<LOG-EXP> == 0 -> PC = PC + 4)  证明：  (1) 当n = 1时，代入命题有：σ(<LOG-EXP>) -> σ(<STA-LIST>) || ~σ(<LOG-EXP>) -> skip。  对引理，当n为1，表示只循环一次，使用CI规则有：(<LOG-EXP> < 0 -> <STA-LIST>) || (<LOG-EXP> > 0 -> <STA-LIST>) || (<LOG-EXP> == 0 -> PC = PC + 4)，  进行取值运算，可得引理语义：σ(<LOG-EXP>) -> σ(<STA-LIST>) || ~σ(<LOG-EXP>) -> skip，  可以得到二者语义一致，故 k = 1时成立。  (2) 假设n = N时，即有命题{σ(<LOG-EXP>) -> σ(<STA-LIST>)} \*\* N || ~σ(<LOG-EXP>) -> skip成立。  (3) 当n = N + 1时，在n = N的基础上，进行一次循环。  若<LOG-EXP> == 0，则PC = PC + 4，结束整个循环，取值运算后得到的语义为：  ~σ(<LOG-EXP>) -> skip。  若<LOG-EXP> != 0，则PC跳到<STA-LIST>的起始位置，继续执行语句序列，取值运算后得到的语义为：  σ(<LOG-EXP>) -> σ(<STA-LIST>)，  把上述语义和(2)中假设运用CI规则，有{σ(<LOG-EXP>) -> σ(<STA-LIST>)} \*\* (N + 1) || ~σ(<LOG-EXP>) -> skip成立。  由(1)、(2)、(3)可知，所求证的命题成立，证毕。 |

# 3 形式验证算法

## 3.1 命题映射算法

命题映射算法的作用是把目标码模式转换为命题的形式，以方便进行后续的推理证明。算法中需要把Power PC指令集中每条指令对应的指称语义作为专用公理输入，逐条遍历输入的目标码模式，把每条目标码转化为对应的指称语义的形式，最终把目标码模式的指称语义表示成命题集的形式输出。命题映射算法的伪代码如表7所示。

表7 命题映射算法

|  |
| --- |
| **Algorithm 1** Proposition Mapping |
| **Input:** ObjectCodePatternSet  **Output:** PropositionSet |
| 1: axiomSet = loadAxiom(denotationalSemanticsFileName)  2: **for** each line in ObjectCodePatternSet **do**  3: lines = line.split(regex)  4: lines = filterOtherCharacter(lines)  5: **if** lines.length == 0 **then**  6: continue  7: **else if** lines.length == 1 **then**  8: add new Proposition(lines) to PropositionSet  9: **else**  10: paras = generateParas(lines)  11: seman = generateSemantic (lines, paras, axiomSet)  12: add new Proposition (lines, paras, seman) to PropositionSet  13: **end if**  14: **end for** |

## 3.2 自动推理算法

自动推理算法是本文提出的形式验证方法的核心。算法以3.1中命题映射算法输出的命题集和一阶逻辑的公理集为前提，根据推理规则推导出一系列新命题，把这些新的命题加入前提中进行后续的证明，最终可以构造出证明序列并得到结论。对于不包含循环的目标码模式命题集可以直接对推导出的证明序列进行取值（σ）操作，从而得到目标码模式命题的语义，与C文法单元的语义比较，可以直接判断出二者的语义是否保持一致。对于包含循环的目标码模式命题集，直接对推导出的证明序列进行取值操作却无法完成语义一致性检验过程，需要引入3.3中的循环交互证明算法才能完成整个过程。命题自动推理算法的伪代码如表8所示。

表8 自动推理算法

|  |
| --- |
| **Algorithm 2** Automatic Derivation |
| **Input:** PropositionSet  **Output:** SemantemeSet |
| 1: **for** each p in PropositionSet **do**  2: **for** each q in PropositionSet **do**  3: **if** p == q **then**  4: continue  5: **end if**  6: newProposition = applyDerivationRuleToTwoPropositions (p, q)  7: **if** newProposition != null **then**  8: add newProposition to PropositionSet  9: **end if**  10: **if** q’s content is empty **then**  11: remove q from PropositionSet  12: **end if**  13: **end for**  14: **end for**  15: **for** each p in PropositionSet **do**  16: s = obtainSemantemeFromProposition (p)  17: add s to SemantemeSet  18: **end for** |

## 3.3 循环交互证明算法

循环交互证明算法的理论基础是限定数学归纳法。算法首先引导用户输入n为1时C文法单元的语义，然后按照循环条件分别为真或假时，分别构造新的命题加入到3.1命题映射算法输出的命题集的一个copy中，调用3.2中的自动推理算法对copy命题集进行推理，从而得到此时目标码模式命题的语义。

把目标码模式命题的语义和用户输入的语义对比。若二者语义不一致，则直接给出形式验证过程出现错误的提醒并退出；若一致，提醒用户输入当n为N时C文法单元的语义，在此基础上再次对源语义集进行一次推理。通过CI规则，把推理出的语义结果加入到n为N时C文法单元的语义中，得到n为N + 1时目标代码模式的语义。把用户输入的n为N时C文法单元的语义中的N使用N + 1替换，比较程序推理出的目标码模式语义和用户输入C文法单元的语义在n为N + 1是否一致，返回判断结果。循环交互证明算法的伪代码如表9所示。

表9 循环交互证明算法

|  |
| --- |
| **Algorithm 3**  **Loop Interactive Proving** |
| **Input:** PropositionSet  **Output:** Flag |
| 1: Flag = true  2: **for** i from 1 to 2 **do**  3: userSemantemeSet = ReadUserInputSemanteme ()  4: copy PropositionSet to cpyPropositionSet  5: add true loop condition Propositionto cpyPropositionSet  6: trueSemantemeSet = AutomaticDerivationAlgorithm(cpyPropositionSet)  7: copy PropositionSet to cpyPropositionSet  8: add false loop condition Propositionto to cpyPropositionSet  9: falseSemantemeSet = AutomaticDerivationAlgorithm(cpyPropositionSet)  10: semantemeSet = trueSemantemeSet || falseSemantemeSet  11: **if** i == 2 **then**  12: semantemeSet = CI (semantemeSet, userSemantemeSet)  13: update n from N to (N + 1) in userSemantemeSet  14: **end if**  15: **if** semantemeSet unequal to userSemantemeSet **then**  16: Flag = false  17: **return** Flag  18: **end if**  19: **end for** |

# 4 结论及展望

本文提出了一种基于安全C子集的编译器形式验证方法，该方法在安全C子集的约束范围内引入了C文法单元的概念，把传统的直接对源代码的整体形式验证转化为了对有限的C文法单元的验证，通过证明C文法单元和目标码模式的语义等价性完成了形式验证过程。该方法极大降低了形式验证的复杂度和耗费的时长的，节省了开发成本。同时，基于本文所提出的形式验证算法开发出的验证工具，可以引导用户通过极少的交互完成对循环结构的证明，整个证明过程符合限定数学归纳法的原理。

未来的工作重心是对C文法单元和目标码模式进行更进一步的精化，同时还需要进一步研究循环交互证明算法，以使算法需要更少的人工交互甚至不需要人工交互就能完成推理过程。

# 5 参考文献

1. RTCA Inc., "RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification", Washington D.C.: RTCA Inc., 1992.
2. RTCA Inc., "RTCA/DO-178C: Software Considerations in Airborne Systems and Equipment Certification", Washington D.C.: RTCA Inc., 2011.
3. Baier C, Katoen J P. Principles of Model Checking[M]. Cambridge: MIT Press,2008.
4. Leroy X. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107-115.
5. Motor Industry Software Reliability Association. MISRA-C: 2004: Guidelines for the Use of the C Language in Critical Systems[M]. MIRA, 2008.
6. Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12: 576-580.
7. Floyd R W. Assigning meanings to programs[C]//Proceedings of Symposium on Applied Mathematics, 1967, 19-31.
8. Leroy X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4): 363-446.
9. Leroy X. Mechanized semantics for compiler verification[M]//Certified Programs and Proofs. Springer Berlin Heidelberg, 2012: 4-6.
10. Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers[C]//ACM SIGPLAN Notices. ACM, 2011, 46(6): 283-294.
11. EREF：A Programmer’s Reference Manual for Freescale Power Architecture Processors[M].Rev.1, 2014，Freescale.
12. Henkin L. On mathematical induction[J]. The American Mathematical Monthly, 1960, 67(4): 323-338.

作者：陈志伟，男，北京航空航天大学计算机学院硕士研究生，研究领域：安全关键软件，论文研究方向：软件形式建模与验证，地址：学院路37号北京航空航天大学新主楼G座519，邮编：100191，手机：13693022036，邮箱：chen476328361@163.com