一种基于安全 C 的编译器形式验证方法

陈志伟 谭宇 马殿富

（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。传统检测编译器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、自动化程序低等问题。本文提出了一种基于安全 C 子集形式文法的编译器验证方法，通过形式文法对应的下推自动机识别出源程序中的 C 文法单元，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 编译器 形式验证 C 文法单元 指称语义 语义一致性

Abstract The compiler is an important development tool, its safety and reliability has important implications for the development of safety critical software. The traditional method of detecting compiler error is a lot of testing, but the test has limitations and it is difficult to achieve full coverage. In recent years, formal verification methods in compiler verification field obtain much attention, but the current formal verification method has proved the presence of high complexity, weak verification, and low automated processes. In this paper, the compiler verification method based on a subset of security C formal grammar is proposed. Through formal grammar corresponding pushdown automata recognizing the C grammar unit in the C source program, we convert the formal verification of the compiler to the verification of a limited number of C grammar units. This paper introduces first-order logic axiom system and special axiom, through the way of theorem proving on this axiom system, completed the semantic consistency verification of the C grammar unit and target code pattern, so as to complete the process of formal verification of the compiler.

Key Words Compiler, Formal verification, C grammar unit, Denotational semantics, Semantic consistency

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。按照对航空器失效状态的影响，将机载软件划分为 A（灾难性的）、B（严重的）、C（较重的）、D（较轻的）和 E（无影响）五个软件等级。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、面向对象编程和形式化方法。在软件开发新技术日新月异的今天，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。

编译器作为安全攸关软件开发过程中的关键工具，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编译过程正确性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对软件系统进行证明。当前，广泛使用的编译器形式验证技术有定理证明[3]和程序检测[4]（Program Check）等。定理证明需要证明的是编译器在整个编译过程中的行为操作，其一般基于高阶逻辑和公理，目前尚不能完全自

动化，需要专业人员参与到证明过程中；程序检测是在编译完成后附加了一个检验过程，验证源代码与目标代码的语义是否一致，但验证过程的自动化却是一项比较困难的工作。

本文提出了一种基于安全 C 子集形式文法的编译器验证方法。通过引入 MISRA-C[5]，即汽车制造业嵌入式 C 编码标准，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，对安全攸关系统中使用的 C 语言进行限制。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，形成了安全 C 子集。安全 C 子集严格要求了编译器的成熟度及稳定性，编译器必须真实地反映源代码的结构和语义，以便编译前后的代码比较和追踪。形式文法是描述一门程序设计语言的方法，安全 C 子集也必须符合其对应的形式文法的约定。美国语言学家 N.乔姆斯基等人建立了形式文法和自动机之间的对应关系，而在语言学中把自动机作为语言的识别器，所以安全 C 子集中形式文法对应的下推自动机可以用来识别源程序，从源程序中识别出的文法单元也必须符合安全 C 子集形式文法的定义。因此，对于源程序编译过程正确性的证明，就可以转化为源程序中包含的文法单元语义保持性的证明，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

最早研究编译器正确性的问题是 John McCarthy 和 James Painter[6]，他们首先完成了对一个简单的从数学表达式到机器语言编译算法的正确性的证明，这项开创性的工作引起了计算机科学研究领域对编译器正确性和可靠性研究的热潮。Susan S[7]等人描述了如何从语言的形式化定义中构造正确的编译器。他们使用 Prolog 语言定义源语言和目标语言的指称语义，通过遍历语言的树结构来构造正确性的证明。Thilo Gaul[8]等人提出了如何把编译器的验证过程同传统的编译器开发流程结合起来, 在减少验证工作的同时使得编译器的开发过程符合软件工程的方法。

近年来，比较具有代表性的是 Leroy X[9~10]领导的 CompCert 项目组所做的工作，他们基于 Coq 辅助定理证明工具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。2011 年，Yang X[11]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 C 编译器中，CompCert 表现非常出色，在其已支持的 C 语言子集中，没有找到任何 wrong-code 错误。

日前，微软公司开发了一种新的编程系统 Spec#[12]，它由 Spec#编程语言、Spec#编译器和 Spec#静态程序验证程序组成。Spec#编程语言是面向对象语言 C#的扩展，它以前置条件、后置条件和对象不变式的形式提供方法定约。Spec#编译器静态地强制保证非空类型，为方法约定和不变式提供运行时检测。Spec#静态程序验证程序从 Spec#程序产生逻辑证明条件。在内部，它使用自动定理证明器分析证明条件来证明程序的正确性或从程序中找到错误。Spec#系统是微软公司的一个尝试，以一种更具成本效益的方式去研发和维护高质量的软件，它对形式验证方法在业界的发展和应用有着重要的意义。

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。

每棵语法子树都是使用下推自动机从源代码中识别出的一种语法结构，安全 C 子集中有多种形式文法，而每种形式文法对应的一种下推自动机，因此在源代码中就存在着多种语法结构，这些语法结构的 C 语言表达形式就是 C 文法单元。源程序的编译过程正确性验证可以等价为对每个 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[13]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：

$$p \rightarrow q, p \vdash q$$

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：

$$p, q \vdash p \wedge q$$

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[14]。一般数学归纳法的逻辑表达式为 P(0)Λ(∀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: ∧	S5, S6, S7, S8, S9, S10, CI

<STA-LIST_2> ∧ .L2: }	

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

前提：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。

由前提有，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，由前提有：

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 newPropositionSet do
3:         newProposition = applyDerivationRuleToTwoPropositions (p, q)
4:         if newProposition != null then
5:             add newProposition to newPropositionSet
6:         end if
7:         if q's content is empty then
8:             remove q from newPropositionSet
9:         end if
10:    end for
11: end for
12: for each p in newPropositionSet do
13:     s = obtainSemantemeFromProposition (p)
14:     add s to SemantemeSet
15: 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 stage ← FIRST, LAST 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 stage == LAST then
12:        semantemeSet = CI (semantemeSet, userSemantemeSet)
13:        update n from N to (N + 1) in userSemantemeSet
14:    end if
15:    if semantemeSet != 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 子集形式文法的编译器验证方法，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] Leroy X. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107-115.

[4] Blum M, Kannan S. Designing programs that check their work[J]. Journal of the ACM (JACM), 1995, 42(1): 269-291.

[5] Motor Industry Software Reliability Association. MISRA-C: 2004: Guidelines for the Use of the C Language in Critical Systems[M]. MIRA, 2008.

[6] McCarthy J, Painter J. Correctness of a compiler for arithmetic expressions[J]. Mathematical aspects of computer science, 1967, 1.

[7] Stepney S, Whitley D, Cooper D, et al. A demonstrably correct compiler[J]. Formal Aspects of Computing, 1991, 3(1): 58-101.

[8] Gaul T, Goos G, Heberle A, et al. An Architecture for Verified Compiler Construction[C]//Joint Modular Languages Conference. 1997, 1996.

[9] Leroy X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4): 363-446.

[10] Leroy X. Mechanized semantics for compiler verification[M]//Certified Programs and Proofs. Springer Berlin Heidelberg, 2012: 4-6.

[11] 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.

[12] Barnett M, Leino K R M, Schulte W. The Spec# programming system: An overview[M]//Construction and analysis of safe, secure, and interoperable smart devices. Springer Berlin Heidelberg, 2004: 49-69.

[13] EREF：A Programmer’s Reference Manual for Freescale Power Architecture Processors[M].Rev.1, 2014，Freescale.

[14] 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