中图分类号：TP 311.5

论 文 编 号 ：10006SY1406108

北京航空航天大學

硕 士 学 位 论 文

安全 C 编译器的构建和形式验证方法的研究与实现

作 者 姓 名　　陈 志 伟

学 科 专 业　　软 件 工 程

指 导 教 师　　马 殿 富　教 授

培 养 院 系　　计 算 机 学 院

Research and Implementation of Secure C Compiler Construction and Formal Verification

A Dissertation Submitted for the Degree of Master

Candidate: Chen Zhiwei

Supervisor: Prof. Ma Dianfu

School of Computer Science & Engineering

Beihang University, Beijing, China

中图分类号：TP311.5

论文编号：10006SY1406108

硕 士 学 位 论 文

安全C编译器的构建和形式验证方法的研究与实现

作者姓名 陈志伟 申请学位级别 工学硕士

指导教师姓名 马殿富 职 称 教授

学科专业 软件工程 研究方向 软件形式建模与验证

学习时间自 年 月 日 起至 年 月 日止

论文提交日期 年 月 日 论文答辩日期 年 月 日

学位授予单位 北京航空航天大学 学位授予日期 年 月 日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期：　　年　　月　　日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期：　　年　　月　　日

指导教师签名：_____ 日期：　　年　　月　　日

摘　　要

机载软件作为安全关键系统的重要构成部分，其安全性和可靠性得到了人们的广泛关注。编译器作为机载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。传统检测编译器错误的方法是进行大量的测试，但软件测试具有一定的局限性，只能证明软件中存在错误却无法发现全部的错误，因此只经过软件测试的编译器是不能达到安全关键领域对编译系统要求的。

近年来，形式化验证方法广泛应用于编译器的开发和验证中。形式化验证方法基于严格的数学理论将软件系统和性质用逻辑方法来规约，通过公理和推理规则组成的形式系统来证明一个软件系统的正确性等。然而，形式验证方法却存在证明复杂度高，自动化程序低等问题，对大规模的编译系统软件进行形式化验证的代价将会非常巨大。虽然现有的辅助证明工具能一定程度上提高编译形式化验证的效率，但要求使用者具有较高的理论基础门槛过高。

针对现有的形式化验证方法在编译验证中存在的各种问题，本文提出了一种基于文法单元和目标码模式的编译语义验证方法，这种方法不是直接验证编译器本身而是验证编译过程的正确性。根据上下文无关文法的独立性，编译过程的正确性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的一致性来实现。文中引入了编译验证公理系统和专用公理，通过形式推理的方式完成了 C 文法单元和目标码模式的语义一致性验证。为了支持本文提出的语义验证方法，在现有的编译技术的基础上提出了安全 C 编译器构建中的关键方法，并给出了系统架构，从而最终完成整个编译验证过程。

关键词：编译器，形式化验证，文法单元，目标码模式，语义一致性

Abstract

Airborne software as an important part of safety-critical system, its safety and reliability has been widespread concern. Compiler is an important tool to achieve the software from design to run on the bridge in the hardware in airborne software development process. The traditional way to detect compiler errors is to perform a large number of tests. However, software testing has certain limitations. It can only prove that the software has errors, but can not prove that the software is error-free, so the software-tested compiler can not achieve compile system requirements of the security critical field.

In recent years, formal verification methods are widely used in the development and validation of compilers. Formal verification methods based on rigorous mathematical theory and nature of the software system are the logical way to specifications, based on axioms and inference rules in the form of a system, to prove the correctness of the software system. However, the formal verification method has the problems of high complexity and low automation procedure, and the cost of formal verification of large-scale compilation system software will be very great. Although the existing supporting tools to a certain extent, improve the efficiency of formal verification, but requires users to have a high theoretical basis and the threshold is too high.

In view of the existing methods of formal verification in the build verification problems, this paper proposes a compiler semantic verification method based on grammar unit and object code pattern. This method does not directly verify the correctness of the compiler itself but verifies the compilation process. According to the independence of the context-free grammar, the correctness of the compilation process can be verified by verifying the semantics of each C-gram unit and the corresponding object code pattern after compiling. In this paper, the compile validation axiom system and a special axiom are introduced to verify the semantic consistency of C grammar unit and object code pattern by means of formal reasoning. In order to support the semantic verification method proposed in this paper, the key methods of constructing secure C compiler are proposed on the basis of existing compiling techniques, and gives the system architecture, and finally completed the process of

compilation and validation.

Key words: Compiler, Formal verification, Grammar unit, Object code pattern, Semantic consistency

目 录

第一章 绪论 1
1.1 研究背景及意义 1
1.2 国内外研究现状 3
1.2.1 编译验证 3
1.2.2 程序检验 4
1.2.3 验证工具 5
1.3 研究目标和研究内容 6
1.4 课题来源 7
1.5 论文的组织结构 7
第二章 形式化验证方法关键技术研究 9
2.1 定理证明技术 9
2.1.1 形式推理方法 9
2.1.2 逻辑公理系统 10
2.1.3 定理证明工具 12
2.2 模型检测技术 12
2.2.1 模型检测基本原理 12
2.2.2 状态空间爆炸问题 13
2.2.3 抽象方法 14
2.3 程序检验技术 14
2.3.1 编译过程的正确性 14
2.3.2 程序检验基本思想 15
2.3.3 程序检验过程 16
2.4 比较 17
第三章 编译形式化验证与安全 C 编译器构建的设计 18
3.1 编译验证问题分析 18
3.2 基于 DO-178C 的编译验证过程 19
3.3 编译形式化验证方法 20
3.3.1 编译验证整体架构 20
3.3.2 编译验证公理系统 22
3.3.3 编译验证形式语义 24

3.3.4 编译验证核心方法 26
3.4 安全 C 编译器构建方法 28
3.4.1 安全 C 编译构建方法架构 28
3.4.2 安全 C 编译构建核心方法 30
3.4.3 层级编码和安全 C 检验 33
3.5 小结 34
第四章 编译形式化验证与安全 C 编译器构建的关键技术 36
4.1 编译形式化验证关键技术 36
4.1.1 文法单元和语义 36
4.1.2 目标码模式和命题 37
4.1.3 编译验证证明方法 40
4.2 编译形式化验证关键算法 44
4.2.1 命题映射算法 44
4.2.2 自动推理算法 45
4.2.3 循环交互证明算法 46
4.3 安全 C 编译器构建关键技术 48
4.3.1 安全 C 词法分析方法 48
4.3.2 文法单元识别方法 51
4.3.3 安全 C 层级编码方法 55
4.4 小结 56
第五章 编译验证工具的设计与实现 57
5.1 编译验证工具架构 57
5.2 编译验证工具实现 58
5.2.1 编译前端模块 58
5.2.2 编译后端模块 62
5.2.3 形式验证模块 64
5.2.4 用户界面模块 67
5.3 系统实验 69
5.3.1 实验环境 69
5.3.2 实验过程与分析 69
5.4 小结 73
结论 74
论文总结 74

工作展望 74
附录 76
参考文献 82
攻读硕士学位期间取得的学术成果 85
致谢 86

图　目

图 1　模型检测的流程 13
图 2　编译正确性图 15
图 3　程序检验过程 16
图 4　编译验证系统开发与验证过程 19
图 5　编译正确性验证架构 20
图 6　文法单元验证架构 21
图 7　编译验证流程 27
图 8　编译验证系统架构 29
图 9　安全 C 检验架构 34
图 10　标识符有限自动机 48
图 11　整数有限自动机 49
图 12　自动机的并操作 50
图 13　<if-statement>文法单元有限自动机 52
图 14　编译验证系统框架 57
图 15　编译前端模块类图 59
图 16　编译后端模块类图 62
图 17　形式验证模块类图 65
图 18　用户界面模块类图 67
图 19　编译验证工具输出 70
图 20　系统运行界面图 72

表　目

表 1　形式化验证技术比较 17
表 2　公理系统的符号定义 22
表 3　编译验证公理系统推理规则 23
表 4　Power PC 汇编指令的指称语义 25
表 5　安全 C 子集文法 30
表 6　层级编码示例 33
表 7　C 文法单元和语义 36
表 8　目标码模式 37
表 9　目标码模式命题 39
表 10　<if-statement>证明过程 41
表 11　<while-statement>证明过程 42
表 12　<while-statement>限定数学归纳法证明过程 43
表 13　命题映射算法 44
表 14　自动推理算法 45
表 15　循环交互证明算法 47
表 16　有限自动机识别算法 49
表 17　<if-statement>文法单元 51
表 18　下推自动机算法 53
表 19　层级编码算法 55
表 20　Lexer 类成员说明 59
表 21　Recognizer 类成员说明 60
表 22　Token 类成员说明 60
表 23　SyntaxUnitNode 类成员说明 61
表 24　SyntaxUnitCollections 类成员说明 61
表 25　Assembler 类成员说明 63
表 26　AssemblerExpression 类成员说明 63
表 27　AssemblerDTO 类成员说明 63
表 28　AssemblerFileHandler 类成员说明 64

表 29　Prover 类成员说明.....65
表 30　PropositionMappingAlgorithm 类成员说明.....66
表 31　DerivationAlgorithm 类成员说明.....66
表 32　LoopInteractiveProvingAlgorithm 类成员说明.....66
表 33　MainWindow 类成员说明 68
表 34　Node 类成员说明.....68
表 35　目标码片段 70
表 36　证明序列片段 71

第一章 绪论

1.1 研究背景及意义

随着计算机应用技术的飞速发展，机载软件已经在航空航天领域中得到了广泛的应用。同时，作为安全关键系统的重要构成部分，机载软件规模不断扩大、内部结构更加复杂、应用环境越来越开放，其安全性和可靠性得到了人们的广泛关注。编译器作为机载软件开发过程中的重要工具，负责将源程序作为输入翻译产生目标程序，是实现软件从设计到在硬件上运行的桥梁。本质上，编译器也是一种大型软件系统，包括许多内部组件、算法和它们之间复杂的交互，构建编译器的过程是一个极其复杂的软件工程实践。

在编译器领域鼎鼎大名的 GCC（GNU Compiler Collection 的简称）是一套由 GNU 开发的编程语言编译器。GCC 原本只能处理 C 语言，后来逐渐扩展到能够支持更多的编程语言，如 C++、Objective-C、Java、Fortran 和 Go 等。目前，GCC 已经成为 Linux 下最重要的编译工具之一。2014 年发布的 GCC5 代码量为 1450 万行，单独处理 C 语言编译部分的代码也已经达到 100 万行左右[1]。

GCC 虽然已经发展得较为成熟，但一直以来都存在着许多错误。GCC 官方为此专门建立了一个网站来列出 GCC 已知的 bugs 并鼓励用户提交 bug 报告，以便这些 bugs 能在后续的版本中得到修复[2]。其它商业编译器也一直存在着许多错误，如 LLVM[3]、Java[4]编译器等。这些编译器中已发现的出于各种原因没被修改的 bugs 和还未被发现的 bugs 将一直存在，因此传统编译器无法满足安全关键领域对编译系统的要求。如何确保编译器编译过程的正确性和构造一套遵循 DO-178B/C 标准的编译系统是进行机载软件开发所面临的重要难题。

DO-178B[5]《机载系统和设备认证中的软件要求》是美国航空无线电委员会（RTCA）于 1992 年颁布的航空适航认证标准，适用于机载系统和设备中软件的开发和合格审定。DO-178B 标准规定了软件开发过程中各阶段要达到的安全性目标，以使软件开发能在满足适航要求的安全性水平下完成其预定功能，但没有规定对其特定的安全性目标需要提供的安全证据以及收集安全证据的技术和方法。RTCA 于 2012 年又发布了 DO-178C[6]。DO-178C 对 DO-178B 的补充有四个方面：形式化方法、基于模型的开发、面向对象编程技术和航空软件开发与验证相关工具。DO-178C 明确了将形式化方法引入到了机载软件甚至 A 级软件的开发和验证中，强调了整个过程的双向追溯性和类型的一致性等。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安全关键系统的开发过程。

近年来，形式化验证方法在编译器的正确性验证中引起了广泛的关注。形式化验证方法基于严格的数学理论将软件系统和性质用逻辑方法来规约，通过公理和推理规则组成的形式系统来证明一个软件系统的正确性。目前，在编译器形式验证领域广泛使用的技术有定理证明、模型检测（model checking）和程序检验（program checking）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，使用公理系统中的推导规则进行推导，目前尚不能完全自动化，需要专业人员参与到证明过程中。模型检测是一种自动形式化验证技术，用于判断计算机系统的行为属性是否正确。模型检测的基本方法是用一个状态迁移图 M 来表示待检测系统的模型，用时序逻辑公式 φ 来描述系统的正确行为属性，通过对模型状态空间的穷举搜索来判断该公式是否能够在模型上被满足，若满足则系统正确性得到证实。模型检测在实际中应用的主要瓶颈是状态空间爆炸问题。程序检验是一种用于确认编译器编译的源代码和生成的目标代码之间语义是否等价的形式化方法，是用统一的语义框架为整个翻译过程的源和目标代码建模，两个模型之间定义一种求精等价关系，还需要设计一个检验器。检验器在编译器每一次运行后形式化地证明生成的目标代码是源代码的一个正确翻译，它不关心编译器的具体实现，只对编译器编译的源代码和目标代码进行处理。

实际中在进行安全关键系统的开发时，除了注意编译系统本身可能引入的错误外，还要避免由于 C 语言本身的缺陷可能产生的问题或者编译器支持的不够完善的地方，需要对 C 语言的使用加以严格的限制，因此我们引入了 MISRA-C 标准[7]。MISRA-C 本是汽车制造业嵌入式 C 编码标准，从 MISRA-C:2004 开始应用范围扩大到其他高安全性系统。MISRA-C:2004 标准共包含 141 条规则，其中强制规则有 121 条，建议规则有 20 条。任何遵循 MISRA-C:2004 标准的 C 代码都应该适合每条强制规则，如果不是，就需要一定的形式化支持，并且在通常情况下都应该遵守 20 条建议规则。使用 MISRA-C:2004 标准也会对程序产生负面的影响，如增大代码规模、降低程序执行效率和影响程序可读性等，所以实际中也需要结合不同领域的软件的特点对 MISRA-C 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 子集提供了安全关键系统的开发中 C 语言的限制集合，通常称为“语言子集”（language subset）。安全 C 子集对编译器支持的 C 语言

标准作出了一定的规定，并要求编译器如实的反映代码的结构和语义，使得对编译前后代码的比较和追踪成为可能。

最后，为了解决编译过程正确性问题并提供对安全 C 子集的支持，本文提出了一种基于文法单元和目标码模式的编译语义验证方法，并且结合文法单元的概念和下推自动机相关理论，设计并实现了一套遵循 DO-178C 的安全关键系统的形式化建模、验证和追踪的编译系统工具集。

1.2 国内外研究现状

1.2.1 编译验证

编译正确性是指编译过程的正确, 即需要确保编译前后的源代码和目标代码在执行时的行为相同，也就是语义保持一致。John McCarthy 和 James Painter 是最早进行编译正确性形式化验证方面研究的学者，他们于 1967 年首先实现了对一个编译算法的正确性的形式化证明。虽然这个算法只能简单地把包含常数、变量和+号的数学表达式翻译为机器语言，但算法的整个形式化证明过程和使用计算机来自动检查证明过程的思想为后续研究者打开了编译验证的大门[8]。

F.Lockwood Morris[9]（1973 年）认为编译正确性问题与无限制的程序正确性问题相比应该被更少的一般化和更好的结构化，为此他对编译正确性进行了一些限定。Morris 给出了第一个明确的编译正确性转换示意图，示意图由源语言和目标语言的语法、语义以及将语法映射到语义的函数组成，转换过程是将源语言语法映射到目标语言语法, 源语言语义对应到目标语言语义。Morris 基于转换示意图完成了一种专门处理类似于 Algol 语法的程序语言的简单编译器的正确性证明，但他也承认在证明过程中忽略掉了一些编译器正确性的数学事实。Morris 创建的转换示意图极大的影响和启发了后续研究者的工作，有力的推动了整个编译领域和形式化验证理论的进步。

1975 年，Laurian M.和 David F.展示了一个将 Floyd 和 Hoare 的归纳断言方法应用于编译正确性问题的工作[10]。Floyd 和 Hoare 的验证方法不需要在程序的表示上做任何更改，但用在编译器的证明上还不够完善。Laurian 和 David 证明了一个适用于简单块结构编程语言的语法制导后缀编译器的正确性，他们把编译器正确性证明划分为两个主要部分：（1）翻译属性的证明，包括源语言的上下文相关特征的实际处理和对语言语法的形成规则使用归纳法；（2）相应的源和目标程序段的语义等价性证明。他们选择了 Floyd 形式语义（Hoare 证明规则）来定义源语言语句的语义和相应的目标代码段语义，并在

此基础上定义了语义等价性关系和编译器正确性模型，通过定理证明的方式完成了整个验证过程。最后，他们指出上述证明方法可以扩展到其它程序语言，但需要满足一些语法限制，即递归过程无参数存在、非递归过程满足霍尔规则等。

1991 年，Susan Stepney 等人提出了另一种验证编译正确性的思路[11]。他们描述了如何从一种语言的形式定义中构造编译器的方法，这种编译器因其构造过程的形式化，故具有正确的属性且容易被证明是正确的。一个被完全证明的编译器自然不会为编译过程引入错误。他们使用指称语义来指定高级语言 Tosca（not a Toy language, for Safety Critical Applications）的语义，利用 Prolog 语言的 DCTG 语法定义这些语义，从而生成了一个解释器。指称语义通过为语言的每个结构指定一个数学值，即“含义”，来定义一个语言，从而允许计算每个程序的抽象机器无关含义。指称语义是在适当的抽象层次构建一个编译器。他们定义了目标语言的指称语义，通过把源语言的操作语义指定为目标语言中的代码模板，计算出代码模板的含义来证明他们与相应源语言结构的含义相同。最后，把操作语言写成 DCTG 语法的形式，从而也完成了编译器的构建工作。

Bahr、Patrick 和 Graham Hutton[12]（2015 年）也开发了一种简单但通用的技术，他们允许通过系统计算的方法从高级语义中派生出正确的编译器，编译器的所有实现细节自然了落在计算过程之外。这种方法是基于标准的等式推理技术并已被应用于计算编译器的大量语言特征及其组合，包括算术表达式、异常、状态、各种形式的 lambda 演算、有界和无界循环、非确定性和中断等。最后，他们所有的程序和计算都是用 Haskell 语言编写的，但只使用基本的递归类型、递归函数和归纳证明，所有的计算过程都已经使用 Coq 辅助证明工具进行了形式化。Coq 可以用作交互式工具来派生出正确的编译器，Coq 不仅能指导用户完成计算过程，还能检查其正确性。如果需要的话，使用 Coq 的代码提取工具可以完全自动地提取编译器和虚拟机实现。

1.2.2 程序检验

程序检验是一种用于验证编译器的源和目标代码之间的语义等价性的形式化方法，它与普通编译过程是一样的，只是在编译完成后附加了一个检验器。1989 年，Blum 和 Kannan 首次正式给出了程序检验的概念并定义了一个程序检验器[13]。检验器本身是一个应用于程序 *P* 的程序 *C*。程序 *P* 运行任何实例 *I* 时，程序 *C* 随后都会运行。程序 *C* 最后或者证明程序 *P* 在实例 *I* 上是正确的，或者声明程序 *P* 是有错误的。他们还为一些特定的、经过仔细选择的问题设计了程序检验器，这些问题都是 P 类问题，即可以在多

项式时间内解决。他们还将现代的密码学方法，特别是概率交互证明的思想，应用在程序检验器的设计中。

1998 年，Amir Pnueli、Michael Siegel 和 Eli Singerman 提出了使用翻译确认的方式来检验编译器的正确性[14]。这种方式是在编译器每次完成编译运行后，增加一个检验阶段以确保生成的目标代码正确的实现了源程序，因此需要一个 Analyzer，也就是程序检验器。他们使用检验器完成了一种同步多时钟数据流语言 SIGNAL 到非同步的 C 代码的翻译过程的检验，并让检验器生成一个检验过程的证明脚本以增加可信度，也方便使用其他工具来检验证明脚本。

2003 年，Sabine Glesner 描述了使用携带证明的程序检验方式来确保编译器的正确性[15]。程序检验方法早已成功应用于编译器前端，但其应用于被优化的编译器后端仍然是一个困难的问题，Glesner 提出使用携带证明的程序检验方法成功的解决了这个难题。他们扩展了检验的场景并要求在检验编译器实现过程中输出一个证书，以告知检验器应该如何计算解决方案。检验器使用此证书来重新计算解决方案，只有当已实现的解与重新计算的解相同，才能证明此种编译器的实现方案是正确的。最后，Glesner 为一个实际的工业项目 AJACS[16]（Applying Java to AutomotiveControl Systems）的代码生成器设计并实现了一个检验器来测试他所提出的方法，实验结果证明了携带证明的程序检验方法可以处理完整的现实编程语言。

1.2.3 验证工具

在形式化验证领域，目前学术界主要使用辅助定理证明工具来完成整个证明过程。下面将主要介绍一下 Coq 工具的相关情况。

Coq 辅助定理证明器[17]是 Thierry Coquand 等人于 1984 年开发出的，是一种基于高阶逻辑的交互式定理证明工具，可以用于验证定理证明是否正确。现如今 Coq 已经发展成了一门语言，在编译器的形式验证中应用得非常广泛。2009 年，Xavier Leroy[18]公布了 CompCert 编译器的开发和验证进展。CompCert 编译器能把函数式编程语言 Clight 的程序转化为基于 PowerPC 指令集的汇编程序，整个过程涉及到了多种不同的中间语言之间的转换，然后使用 Coq 证明工具对完整的编译链的端到端进行了语义可保持性验证。Leroy 最后指出了整个 CompCert 项目依旧在进行中，许多的工作还需要被完成，如处理一个更大的 C 语言子集部署并证明更多优化方法，把语义一致性证明扩展到共享内存的并发中等。

2011 年，Yang X[19]等人在关于 Csmith 的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。他们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，没有找到任何代码错误。CompCert 项目迄今获得的初步结果有力的证明了使用现有的有限的辅助证明工具、基本语义和算法可以实现形式上验证真实编译器的目标。

2016 年，Leonardo Rodríguez[20]等人证明了一个按名调用函数式编程语言编译器的正确性，证明方法是基于这种语言领域理论的指称语义。源语言是带有递归的简单类型的 lambda 演算的扩展，目标语言是 Krivine 抽象机[21]的扩展。他们使用步进索引的逻辑关系和双正交性，以组合的方式获得正确性的概念。这种抽象设置在运行环境的修改方面提供了一定程度的灵活性，并且相对于语言的构造函数也是模块化的。步进索引的使用使他们能够在递归的存在下处理归纳证明。他们所有的结果都使用 Coq 辅助证明工具进行了形式化证明。最后，Rodríguez 指出他们未来的工作是计划通过丰富类型系统和添加新的构造函数来扩展源语言，并把他们的方法应用到其它更接近实际汇编代码的执行模型中。

1.3 研究目标和研究内容

本文的研究目标是提出一种基于语义的形式验证方法来保证编译过程的正确性，并实现一个符合 DO-178C 规范的集建模、验证和追踪于一体的编译验证工具原型。该工具不仅能完成基本的编译功能，如词法分析等，还可以检查源代码是否符合安全 C 标准；能够实现本文提出的形式验证方法对整个编译过程进行验证且能生成正确的目标代码；能够从源代码追溯到目标代码，实现编译过程的完整性、一致性和准确性的需求；能够实时反馈编译和验证过程的信息。主要的研究内容有：

(1) 研究如何在编译阶段加入对安全 C 规则的检验过程，使不符合安全 C 规则的源代码在初始阶段就能被识别出，并修改现有的编译器构造方法使其能支持本文提出的形式验证方法。

(2) 研究一种基于编译语义的形式验证方法检验编译过程是否正确。由形式文法和自动机的相关理论，本文把对源程序编译过程正确性的证明转化为对源程序中包含的文法单元和目标码模式的语义一致性证明。通过设计命题映射算法把文法单元对应的目标码模式转化为命题，又基于编译验证公理系统设计命题自动推理算法，从公理系统中事

先给定的公理出发，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。比较最终推导出来的证明序列的语义与前置条件的语义是否一致，从而完成整个证明过程。

(3) 针对 A 级软件开发中源代码和目标代码的可追踪性需求，设计一种方法实现源代码中的每一个语句与汇编代码相应片段的对应。

(4) 基于上述方法的编译验证工具的设计与实现。编译验证工具需要能自动识别出输入的源代码中包含的文法单元，对于普通的运算、赋值语句等由于其语义较简单，只需要保证其语法正确；对于循环和选择语句不仅要保证其语法正确，还要使用形式验证工具保证其语义的一致性。

1.4 课题来源

本课题来自民机专项“符合 DO-178B/C 的 A 级机载软件开发与认证技术研究”。

1.5 论文的组织结构

第一章介绍了本文的研究背景及意义，并对国内外关于编译正确性方面的研究工作做了总结和分析，明确了本文的研究目标和研究内容。

第二章介绍了编译形式化验证的相关技术，包括定理证明技术、模型检测技术和程序检验技术，并对这些技术的优势和面临的问题作出了详细的分析和总结。为后文编译形式化验证和工具的实现提供了理论和技术基础。

第三章给出了基于 DO-178C 规范的编译验证系统开发与验证过程，提出了一种基于语义的编译形式化验证方法，详细阐述了整个证明思路。同时，为了支持编译语义验证方法，在现有的编译技术的基础上提出了安全 C 编译器构建中的关键方法，并给出了系统的整体架构。

第四章介绍了编译形式化验证与安全 C 编译器构建的关键技术，引入了文法单元和语义、目标码模式和命题等核心概念，提出了编译形式化验证过程中的三大关键算法。研究了如何把有限自动机和下推自动机方法应用到安全 C 编译器的构建和验证中，并提出了层级编码算法，实现了安全 C 编译的可追踪性需求。

第五章在前文的基础上，本章详细讨论了编译验证工具的具体实现，首先给出了工具的系统架构图和模块划分，然后从编译前端模块、编译后端模块、形式验证模块和用户界面模块四个部分详细描述了系统的具体实现。最后，设计了一个测试样例对实现的

工具进行实验，所得到的实验结果证明了本文提出的编译形式化验证方法和安全 C 编译构建技术的有效性和科学性。

结论部分总结了全文的主要工作，分析了本文取得的研究成果和存在的问题，并对未来工作的方向进行了分析和展望。

第二章 形式化验证方法关键技术研究

本章对现有的与编译相关的形式化验证技术进行了研究和总结，其中包括定理证明、模型检测和程序检验三种形式化技术，并分析和比较了这些技术的优劣性。本部分为本文的研究和技术实现提供了理论和技术基础。

2.1 定理证明技术

2.1.1 形式推理方法

形式推理是指从一般性的前提出发，通过推导即“演绎”而得出具体陈述或个别结论的过程，是定理证明方法的核心。

假设需要证明的某个目标为 P，而这个目标依赖于某些假设 HYP（也称为前提或断言），在证明 P 时可以先假定 HYP 成立，则可以说目标 P 是在假设 HYP 之下证明的，也可以说由假设 HYP 可以推演出结论 P。这一情况可形式化地表示如下：

$$HYP \vdash P$$

这种形式语句称为一个推演式。在这种推演式里，集合 HYP 里的每个假设以及目标 P 都是称为谓词的公式。一个谓词就是一个形式化的语句，它描述了我们可能要假定的某个性质，或者我们希望证明的某个性质。

为了得到一个推演式，大部分时候都需要隐式的使用一些规则，借助于它们将不同推演式联系起来，这种规则称为推理规则。存在某些更为基本的推理规则，它们完全不依赖任何特殊的数学领域，是证明推演式时某些最基本东西的形式化。下面将介绍 4 条最基本的推理规则。

第一条推理规则形式化了推导出的概念，它实际上是一个公理，给出了关于假设概念是永远正确的事实。给定谓词 P，P 总可以在假设 P 本身的情况下证明。可以形式化地写为：

$$\frac{}{HYP \vdash P}$$

第二条推理规则讨论的是相对于假设集合的证明的单调性。给一个推演式增加假设不会破坏已经得到的有关这个推演式的证明，新增加的假设在新证明中仅仅扮演着某种不活动的角色。可以形式化地写为：

$$\frac{HYP \vdash P \quad HYP\text{包含于}HYP'}{HYP' \vdash P}$$

第三条推理规则很容易从前面两条规则中推导出，含义是谓词 P 是假设集合 HYP 中的一个假设。可以形式化地写为：

$$\frac{P\text{出现于}HYP}{HYP \vdash P}$$

第四条推理规则说的是另一个事实，如果我们证明了一个形式为 $HYP \vdash P$ 的推演式，而后将结论 P 用做假设证明了另一个形式为 $HYP, P \vdash Q$ 的推演式，这时就可以断言，现在已经有了对推演式 $HYP \vdash Q$ 的一个证明。可以形式化地写为：

$$\frac{HYP \vdash P \quad HYP, P \vdash Q}{HYP \vdash Q}$$

有了推理规则后，对于一个确定的推演式 S 的形式证明就能以一种很形式化地方式进行。设 $S1, S2, ..., Sn$ 和 S 都是推演式，也就是具有 $HYP \vdash P$ 的形式，则有：

$$\frac{S1 \quad S2 \quad ... \quad Sn}{S}$$

推演式 $S1, S2, ..., Sn$ 称为规则的前件，而推演式 S 称为后件。这一规则表示有关 S 的证明可以规约到 $S1, S2, ..., Sn$ 的证明，即为了证明结论 S，只需证明所有的前件就够了。实际中，一旦完成整个证明过程，推演式 S 就被称为是一个定理，最后可以把 S 加入规则集合中。可以发现，对一个推演式的证明是相对于一个特定的推理规则集合完成的。

2.1.2 逻辑公理系统

逻辑公理系统是定理证明的基础。20 世纪 60 年代 Hoare 和 Floyd[22, 23]首先在他们的论文里提出一个形式系统，称作霍尔逻辑系统。霍尔逻辑描述程序正确性的一般形式为：$\{Pre\}P\{Post\}$。其中，Pre 称为前置断言，$Post$ 称为后置断言，P 为程序代码。若 P 的每一次计算开始于满足 Pre 的状态，执行终止时的状态满足 $Post$，则正确性公式为

真，程序 P 具有完全的正确性。在霍尔逻辑中存在一组证明规则，称为霍尔规则。这些规则把证明一条复合命令的部分正确性断言简化为证明它的直接子命令的部分正确性断言。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[24, 25]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，如对并发程序和资源管理的验证，其有望成为程序形式验证的又一种重要方法。

分离逻辑对程序的推理也是采用霍尔三元组 $\{P\}C\{Q\}**$，其中，P 表示前置条件，Q 表示后置条件，而 C 表示代码段。在分离逻辑中，前置条件和后置条件中的程序状态主要由栈 s 和堆 h 构成，栈是变量到值的映射，而堆是有限的地址集合到值的映射。在程序验证时，可以将栈看作对寄存器内容的描述，而堆是对可寻址内存内容的描述。分离逻辑中引入了两个新的分离逻辑连接词：分离合取 $*$ 和分离蕴含 $-\!\!*$。$[P*Q]sh$ 表示整个堆 h 被分成两个不相交的部分 h_0 和 h_1，并且对子堆 h_0 断言 P 成立，而对子堆 h_1 断言 Q 成立。形式化表示如下：

$$[P*Q]sh \overset{def}{=} \exists h_0\, h_1 . h_0 \perp h_1 \wedge h_0.h_1 = h \wedge P\, s\, h_0 \wedge Q\, s\, h_1$$

其中 $h_0 \perp h_1$ 表示堆 h_0 和 h_1 不相交，$h_0.h_1$ 表示堆 h_0 和 h_1 的联合。

$[P-\!\!*Q]sh$ 表示如果当前堆 h 通过一个分离的部分 h' 扩展，并且对 h' 断言 P 成立，则对扩展后的堆 $(h.h')$ 断言 Q 成立。形式化表示如下：

$$[P-\!\!*Q]sh \overset{def}{=} \forall h'.(h' \perp h \; and \; P\, s\, h') \rightarrow Q\, s\, (h.h')$$

分离逻辑因其本身所蕴含的分离思想，在验证程序时能够简洁、优雅地支持进行局部推理和模块化推理，已经在程序验证领域得到了重视和广泛使用。但是，将分离逻辑用于解决更复杂的软件系统的源代码级验证仍然需要进一步解决许多技术难题，如对语言类型的支持范围、验证过程的自动化程度等。

2.1.3 定理证明工具

目前业界主流的定理证明方式是使用辅助定理证明工具，如 PVS、Coq 和 Isabelle 等，但不足之处在于这些工具都需要人工进行交互。

PVS[26]是原型验证系统（Prototype Verification System）的缩写。该系统不仅包括规约语言和定理证明器两部分，还集成了解释器、类型检查器及预定义的规约库和各种工具等。PVS 提供的规约语言基于高阶逻辑，并且具有丰富的类型系统，是一般适用的语言，表达能力很强。大多数数学概念、计算概念均可用该语言自然直接地表示出来。PVS 定理证明器以交互的方式工作，同时又具备高度的自动化水准，为计算机科学中严格、高效地应用形式化方法提供了自动化的工具支持。

Coq[27]是一个交互式定理证明器，提供了一种类似书写数学定义的形式语言、可执行算法以及用于机器证明检查的半交互式开发的环境。Coq 包含自动定理证明策略和各种决策的程序，它允许用户定义自己证明方法和策略的语言。Coq 实现了一种程序规范和名为 Gallina 数学高级语言，Gallina 基于归纳建构的微积分表达形式，其本身是一种结合了高级逻辑和丰富类型的函数式编程语言。通过命令的本地语言，Coq 允许定义有效评估函数或谓词，表述数学定理和软件规范，交互地开发这些定理的形式证明，使用相对较小的认证“内核”对这些证明进行机器检查，以及将已认证的程序提取到 Objective Caml、Haskell 或 Scheme 等语言。总之，Coq 是一种基于高阶逻辑的用于验证定理的证明是否正确的形式化工具。

Isabelle[28]是一种通用的定理证明器，它为证明系统的开发提供了一个通用的框架。Isabelle 以人机交互的形式实现定理证明，并通过应用策略和策略组来支持自动证明，其中高层的证明由人来进行控制，底层的简单证明由机器来自动完成。Isabelle 支持对数学公式的形式化描述，并为这些公式的逻辑演算提供了证明工具，支持多种对象逻辑，如高阶逻辑（HOL）、模态逻辑（ML）等，允许自定义新的逻辑，还可以通过定义对象逻辑具体和抽象的句法以及推理规则来实现一个新的逻辑系统。Isabelle 具有丰富的类型系统，包括元组类型、函数类型和多态类型等，有强大的规则库和灵活高效的命令集，支持前向证明和后向证明这两种验证方式。Isabelle 在计算机硬件和软件，以及计算机语言和协议属性的形式化验证中应用得十分广泛。

2.2 模型检测技术

2.2.1 模型检测基本原理

模型检测[29, 30]是一种验证给定系统是否满足给定待测属性的形式化技术。编译器可以使用模型检验技术对所编译的对象进行验证。模型检验工具还可以利用编译器对程序的精确分析来优化模型的状态空间。

模型检测的基本方法是用一个状态迁移图 M 来表示所要检测系统的模型，并用模态/时序逻辑公式 ϕ 来描述系统的正确行为属性，然后通过对模型状态空间穷举搜索来判断该公式是否能够在模型上被满足。如果公式在模型上满足，即 $M \vDash \phi$，则系统的正确性得到证实；反之，表明系统中存在错误，即 $M \vDash \sim\phi$，则系统正确性被证伪。模态/时序逻辑是模型检测的基础。常用的模态逻辑有三种，即计算树逻辑[31]、线性时序逻辑[32]和命题 μ 演算逻辑[33]。

模型检测的一般流程是：首先对待检验的系统进行建模，然后使用一种形式化的语言如时序逻辑表达式来描述系统属性，最后使用相应的模型检测分析技术来判断此系统模型是否满足系统属性。模型检测的流程如图 1 所示。

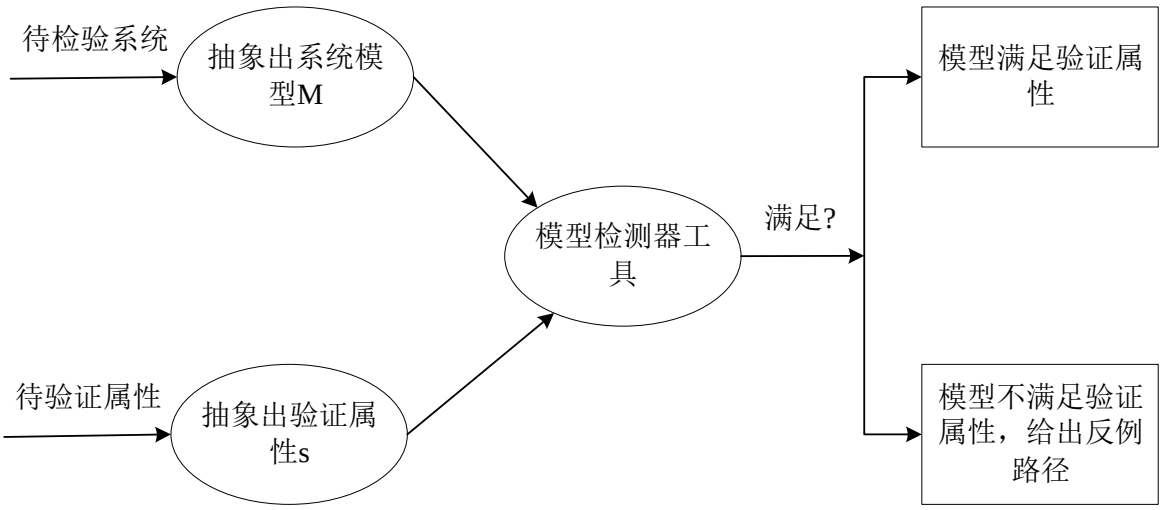

图 1　模型检测的流程

图 1 中，模型检测一般包括以下三部分：（1）描述系统的建模语言；（2）描述系统属性的说明语言；（3）一项验证系统满足正确性需求的分析技术。其中，这里待检验的系统模型一般使用转换系统（Kripke 结构）或者自动机来描述，而系统的规范、属性则使用时态逻辑表达式或者自动机来表示。

2.2.2 状态空间爆炸问题

状态空间爆炸问题是模型检测在实际应用中的主要瓶颈。由于模型检测基于穷举搜索来对正确属性进行判断，所以它适用于对有限的、状态空间较小的系统进行分析。然

而，在实际应用中经常存在着状态空间庞大的系统，如对于软件系统，由于存在着无限数据域（如整数）、无界数据类型（如链表）、以及复杂的控制结构（如递归），导致软件系统的状态空间可以是无限大，直接对它们进行模型检测在实际中是不可行的。状态空间爆炸问题已成为模型检测方法应用于软件评价及验证的一个具有挑战性同时又无法回避的难题。

对于这一难题人们提出了很多减少和压缩状态空间的方法，如符号模型检测、对称模型检测、程序切片和抽象等。在众多的状态减少和压缩技术中，抽象技术是解决状态爆炸问题的最有潜力的方法之一，同时抽象也是有效集成模型检测和定理证明的方法，通过抽象技术可以实现两种形式化方法的互补。

2.2.3 抽象方法

抽象方法是解决状态爆炸问题的一个重要的方法。它的基本思想是首先构造一个比原系统的具体模型小的有限抽象模型，然后通过正确属性在抽象模型上的检测结果推测出其在原来的具体模型上是否可满足。抽象方法依据对于所给定的某种正确属性而言，待检测的原系统的许多信息是无关的，如某些程序变量的取值、进程的标识符、调用栈中活动记录等。因此，这些信息可以从具体模型中抽象出去。这样不仅简化了模型，同时保留了必要的信息，使得抽象的模型检测得以有效地进行。

模型检验技术已经在各种协议和硬件的设计及检验上取得了成功，但因为软件程序代码的描述比较复杂，包含的状态空间通常可以是无限的，所以验证难度较大。程序代码的模型检验一般按照“抽象—细化—验证”的步骤进行，即将程序代码中变量的取值情况作为状态划分的依据，从初始状态的制定开始根据状态转换关系生成状态转换路径，代表代码的执行情况；把被验证属性表示为状态转换路径必须满足的具体要求，如果状态转换路径满足这种要求，则代码满足被验证属性，验证结束；否则，代码不满足被验证属性，则生成反例路径证明被验证属性如何被破坏。

模型检测的优点在于可以完全自动地进行验证，但在实际应用中也需要更多软件工具的支持，业界比较著名的有 SMV[34]、SPIN[35]和 CWB[36]等。

2.3 程序检验技术

2.3.1 编译过程的正确性

编译正确性是指编译过程的正确性。编译过程正确性是要保证输入的源代码和编译

后的目标代码语义一致，即源代码与编译后的代码行为上要等价，编译器不能在目标代码中改变源代码中的操作。编译过程正确性的形式化定义可用如下图 2 的转换示意图表示，对编译过程正确性形式化的验证就是证明对应的转换示意图的成立[37]。

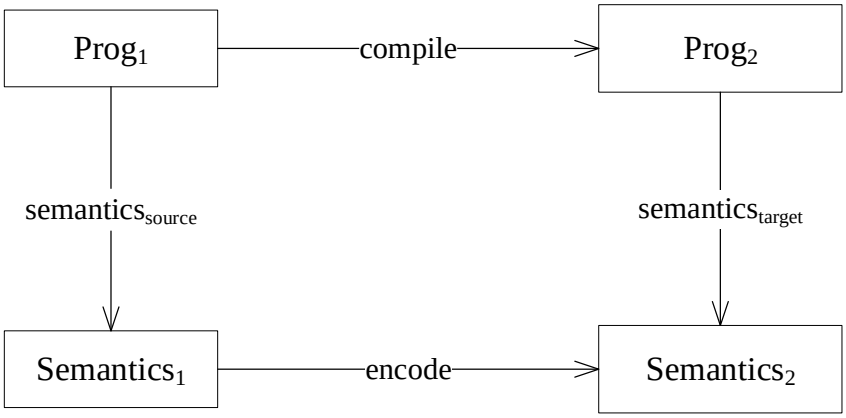

图 2　编译正确性图

图 2 中的箭头可看成是函数映射过程。又由图 2 中上下两条转换路线最终的目的地是相同的，则以下等式是成立的：

$$\begin{cases} \text{Semantics1} = \text{semantics}_{\text{source}}\left(\text{Prog1}\right) \\ \text{Semantics2} = \text{encode}\left(\text{Semantics1}\right) \\ \text{Prog2} = \text{compile}\left(\text{Prog1}\right) \\ \text{Semantics2} = \text{semantics}_{\text{target}}\left(\text{Prog2}\right) \end{cases}$$

化简上述等式，并用 P 代替 $Prog1$，则编译正确性可以用如下等式来表示：

$$encode\left(semantic_{source}\left(P\right)\right) = semantics_{target}\left(compile\left(P\right)\right)$$

程序 P 可以使用不同的语义来解释，如操作语义、公理语义、指称语义等，不同的语义代表着不同的证明方法，但是将图中赋予结点和箭头的不同含义进行抽象，则所有的图示结构都是相同的。也就是说，它们的组成是相同的，均由源语言和目标语言的语法、语义以及将语法映射到语义的函数组成，编译转换过程将源语言语法映射到目标语言语法，源语言和目标语言的语义对应。不同的方法之间的区别只是定义上述组成部分的方式和箭头的方向。图 2 证明编译过程的正确性具有通用性，但是如何将源程序与编译后目标程序的语义对应起来才是证明的关键问题。

2.3.2 程序检验基本思想

程序检验[38, 39]是一种用于检验编译器或代码生成器的源和目标语言之间的语义等价性的形式化方法，它通过证明源代码和目标代码的语义等价性来证明编译器的正确性。使用程序检验方法不是直接验证翻译程序，而是用统一的语义框架为某一翻译过程的源

和目标代码建模，两个模型之间定义一种求精（refining）等价关系，因此需要设计一个检验器，检验器在编译器每一次运行后形式化地证明生成的目标代码是源代码的一个正确翻译。检验器不关心编译器的具体实现，只对编译器的源代码和目标代码进行处理，如果验证成功则编译继续进行，如果发现语义矛盾之处则输出一个警报或取消编译。

编译器的形式化验证可以弱化为对检验器进行形式化验证工作。相对于编译器而言，检验器的形式化验证工作是比较简单的，从而大大减轻了编译验证的难度及工作量。同时，由于检验器不关心编译器的具体实现，所以不会对编译器的设计进行限制，方便了未来对编译器的优化与完善等。

2.3.3 程序检验过程

一个自动化的检验器应该包括以下要素：（1）一个用于描述源语言和目标语言的公共语义框架；（2）基于公共语义框架形式化地建立目标代码和源代码之间的“正确执行”定理；（3）一个有效的证明方法，其中证明代表生成目标代码的一个语义框架模型，正确的实现代表源代码的另一个模型；（4）通过 Analyzer 自动执行证明方法，如果执行成功，则生成一个证明脚本；（5）一个证明检查器，用于对 Analyzer 产生的证明脚本进行检查。

程序检验的过程如下图 3 所示：

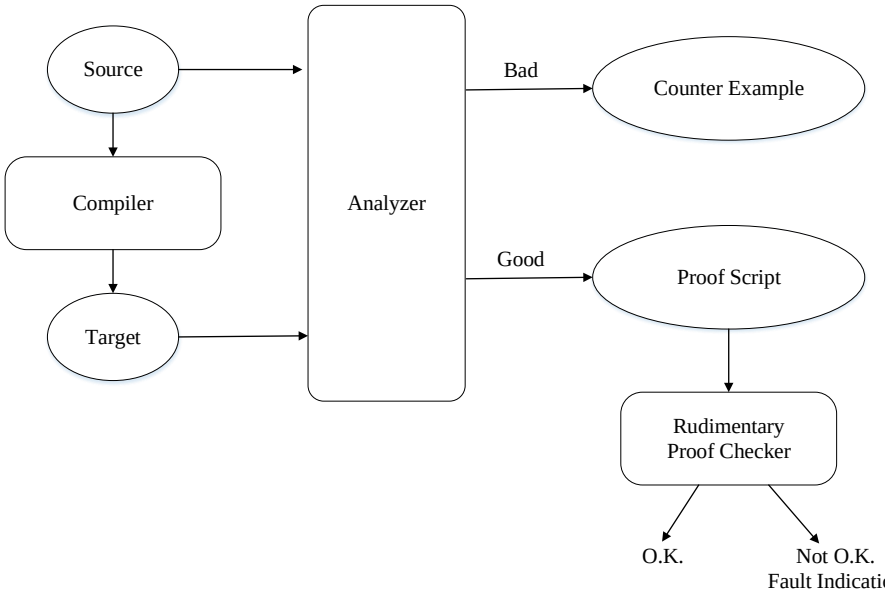

图 3　程序检验过程

分析器接收源程序和目标程序作为输入。如果分析器发现生成的目标程序正确的实现了源程序，则会产生一个详细的证明脚本。如果分析器无法建立源程序和目标程序之间的正确对应关系，它会产生一个反例。该反例包括了生成的目标程序行为不同于源程

序的情景。因此，该反例提供的证据表明编译器有故障，需要加以修改。

2.4 比较

本章我们对三种与编译相关的形式化验证技术进行了介绍并分析了它们的原理，其对比分析如表 1 所示。

表 1　形式化验证技术比较

技术	特点	阶段	不足
定理证明	完全的形式化验证，不会有任何错误；能够使用证明工具，如 coq 等，对编译器进行分阶段	编译器构造时	只能半自动化，需要人来交互；会影响编译器的具体实现和性能；证明难度大
模型检测	完全的自动化验证；能在系统模型出错时给出反例，便于后续追踪和修改	编译运行时	受到无穷状态空间问题的限制；验证过程较为困难
程序检验	与编译器的具体实现无关；不影响编译器的性能；验证简单且工作量低	编译运行后	存在一定概率的误报率，还需要其它证明工具的辅助

定理证明技术是一种以软件系统为公理获得其性质的证明过程。形式推理是定理证明技术的核心方法，霍尔逻辑和分离逻辑是编译正确性验证的两种公理体系。定理证明技术能够基于无穷域上的归纳法处理无穷状态空间问题，但它在自动化程度比较差，需要人工交互，而且也不能在证明失败后提供易于理解的的反例。

模型检测技术是为待检测的系统建立一个用状态迁移图表示的模型，通过对模型状态空间穷举搜索来判断待测属性是否能够在模型上被满足。但是，对编译器这样的系统软件，因其程序代码的复杂性，包含的状态可以空间是无限的，所以会导致状态空间爆炸问题。模型检测的优势是它的自动化程度比较高，并且当系统不能具备所期望的性质时，能给出相应的反例路径，但其验证过程较为困难。

程序检验技术是通过证明源代码和编译后目标代码的语义等价性来完成编译正确性的验证。它在编译器后附加一个检验器，对编译器的具体实现不敏感。同时，检验器是在编译器每一次运行后再运行，不会影响编译器的性能。程序检验技术最大的问题是它的准确率或者说误报率，虽然非常低，但也不能忽视。

第三章 编译形式化验证与安全 C 编译器构建的设计

3.1 编译验证问题分析

认证到 A 级的安全关键系统必须由符合 A 级标准的编译系统产生可执行代码，因此具备高安全性的编译系统是安全关键系统实现的重要保证。A 级软件开发标准要求软件的错误率在 10^{-9} 以下，DO-178C 提出了一套符合 A 级标准的开发方法。DO-178C 规定软件生命周期过程包括软件计划、软件开发和软件验证三个过程，并明确将形式化方法引入到软件的开发和验证过程中。

传统的编译器，如 GCC、LLVM 等，主要使用软件测试的方法来检验编译器的正确性，但软件测试具有一定的局限性，只能证明软件中存在错误却无法发现全部的错误，因此只经过软件测试的编译器是不符合安全关键领域对编译系统要求的。使用形式化方法进行编译系统的开发和验证是目前主流的方式，如借助 Coq 工具使用定理证明方法对编译器的整个构建过程进行验证，从而得到一个经过完全形式化验证的编译器，这是一种直接对编译器本身进行验证的方法；在现有的编译器后面附加一个程序检验器，只验证编译前后的源和目标码的语义是否一致，即编译过程的正确性，而不验证编译器本身等。虽然这些现有的验证方法能确保编译器在一定程度上的高安全性，但也存在着各种问题，如定理证明方法的验证结果虽然安全可靠，但是自动化程度较低；程序检验的方法虽然自动化程度高，但是建模和验证的过程较为困难等。若能合理的去掉这些现有形式化验证方法的短处而取其长处，则可以设计出一种新的满足编译系统高安全性需求的形式验证方法。

针对上述分析，本文提出了一种基于文法单元和目标码模式的编译语义验证方法，该方法不是直接验证编译器本身，而是和程序检验的思路一致验证源和目标码的语义是否一致。程序检验方法是对源和目标码进行整体的建模和证明，但当源代码规模增大、结构变得复杂时，形式验证的难度也急剧增大。本文提出的语义验证方法却能把源和目标码之间整体形式化验证转化为源代码中每一个文法单元单独进行，这种方法的理论基础是上下文无关文法的独立性。源代码规模的增大对应着本文验证方法的文法单元数量的增加，源代码结构的复杂化对应着更为有效的文法单元识别算法等，这些都不会对每个文法单元的形式验证造成影响。本文基于构建的编译验证公理系统，对每个文法单元使用定理证明方法中的形式推理完成了整个验证过程。

3.2 基于 DO-178C 的编译验证过程

根据安全关键领域编译规范以及 A 级软件开发和验证活动的过程和目标，本文提出的基于形式化方法的编译验证系统的开发与验证过程也必须符合 DO-178C 规定的 A 级软件开发和验证标准，且需要支持安全关键软件需求一致性、完整性、有效性和对层次需求的可追溯性验证。下面给出基于 DO-178C 规范的编译验证系统开发与验证过程，如图 4 所示。

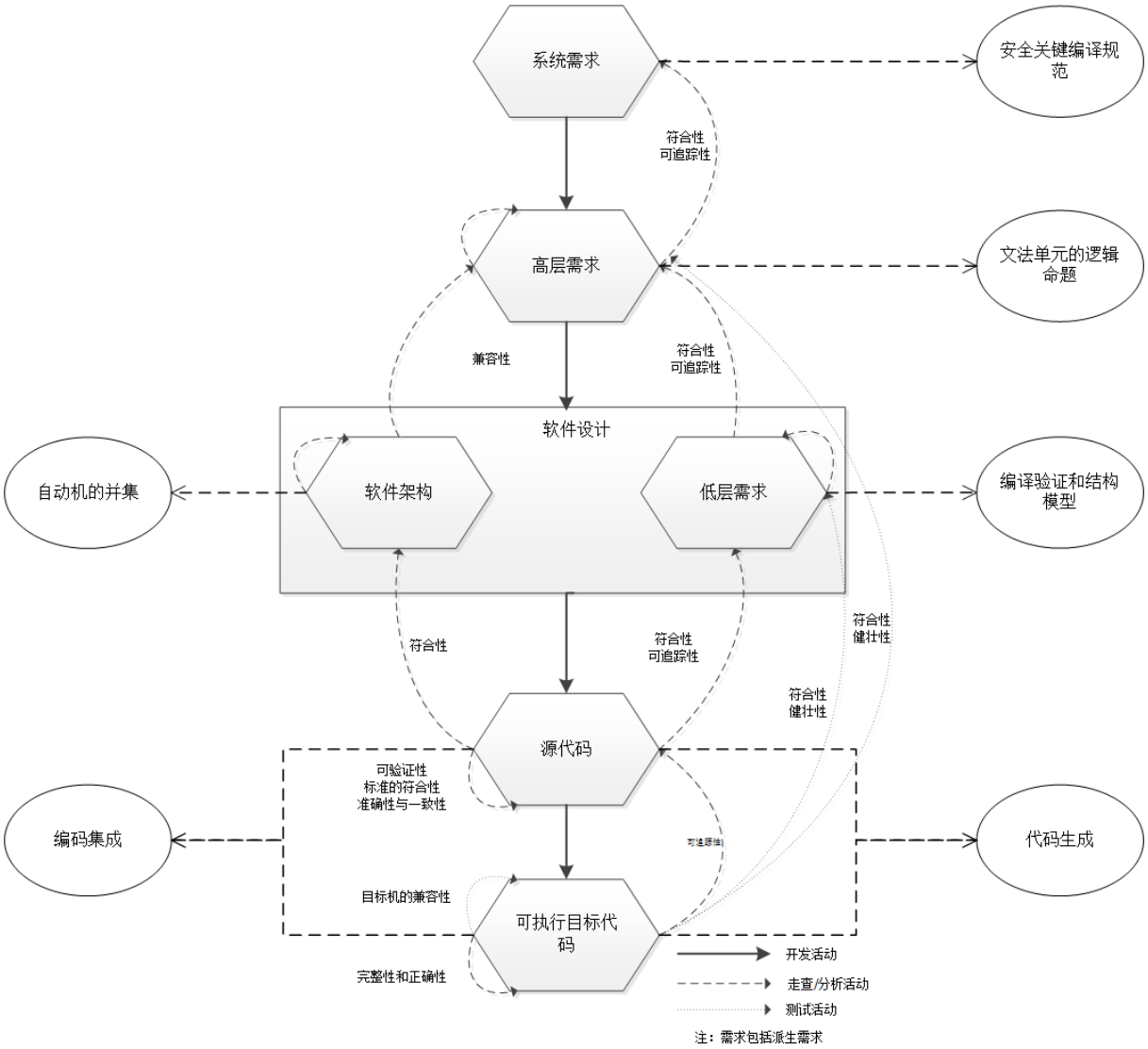

图 4　编译验证系统开发与验证过程

图 4 中，编译验证系统的系统需求为安全关键编译规范，也就是安全 C 子集规范。高层需求为文法单元的逻辑命题，其来源于安全 C 子集的文法。低层需求为编译验证和结构模型，它是高层需求的具体实现，主要包括本文提出的形式验证方法和基于文法单元的编译系统构建。软件架构为自动机的并集，也是上下文无关文法的并集，因为上下

文无关文法和自动机之间存在着对应的关系。软件编码与集成是根据自动机的并集和编译验证结构模型把识别文法单元自动机的实现代码和编译验证代码集成为系统。本文主要研究编译形式验证方法和安全 C 编译系统的构建两个方面，并基于 DO-178C 标准规定的软件开发过程完成了整个编译验证系统原型的实现。

3.3 编译形式化验证方法

3.3.1 编译验证整体架构

编译验证主要有两种方式：一种是验证编译器自身，即验证编译器的实现是否含有错误代码，会不会篡改源代码的语义；另一种是验证编译过程，即验证编译器产生的目标代码和源代码的语义是否一致。编译器自身的正确性是前提，而编译过程的正确性是最终的目的。编译器自身的验证是通过形式化验证方法来保证编译器在构造实现中没有引入任何错误，不会对编译过程产生任何影响，从而间接的保证了编译过程的正确性。验证编译过程的正确性是本文的研究目标，下面将给出编译正确性验证的整体架构，如图 5 所示。

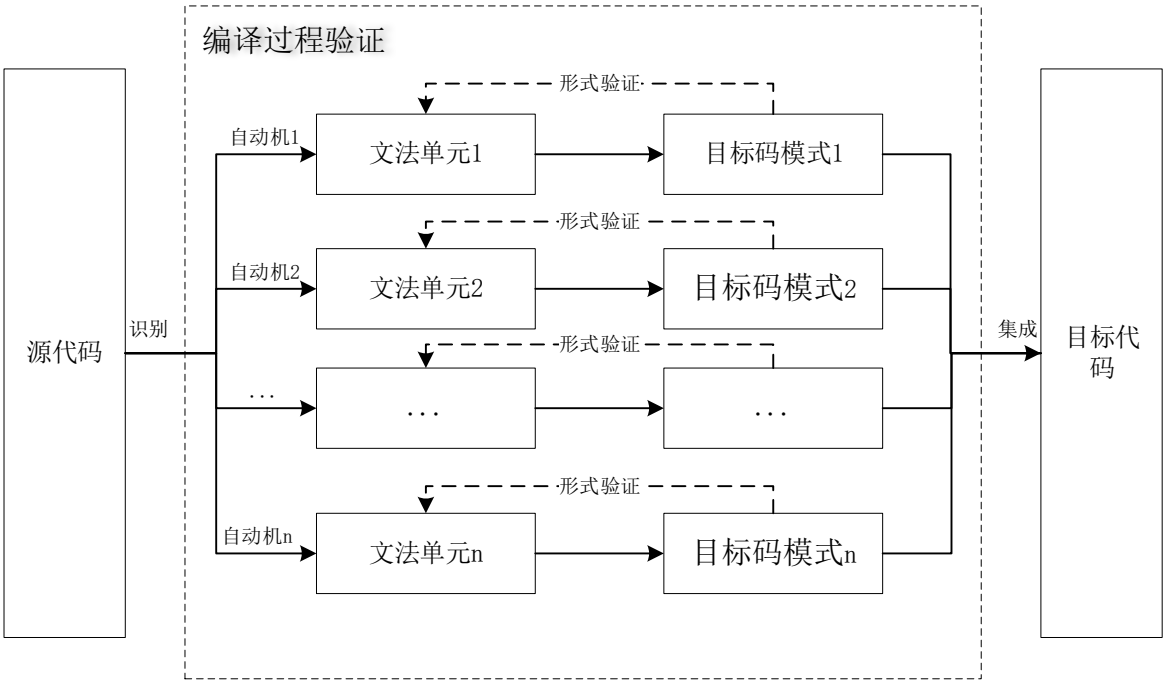

图 5　编译正确性验证架构

图 5 中，编译正确性验证需要先基于安全 C 子集的文法设计对应的下推自动机从源代码中识别出不同的文法单元，并用一种类似于树型的集合结构维持不同文法单元之间的上下文结构关系。根据自动机识别出的文法单元生成对应的目标码模式，这个目标码模式是没有被优化过的且比较固定的。对目标码模式使用形式验证方法进行自动推理并

对得到的结果进行语义的获取和确认过程，即可得到每个文法单元和生成的目标码模式的语义是否保持一致。只有当所有目标码模式的形式验证都通过时，才能证明整个编译过程的正确性并把目标码模式集成为目标代码。

从编译正确性验证的整体架构中可以看到，验证方法的本质是把对编译过程的验证转化为对不同文法单元和对应目标码模式的语义验证，这一过程相对于直接验证源代码和目标代码的语义简单了很多，效率也比较高。下面将给出文法单元验证的架构，如图6所示。

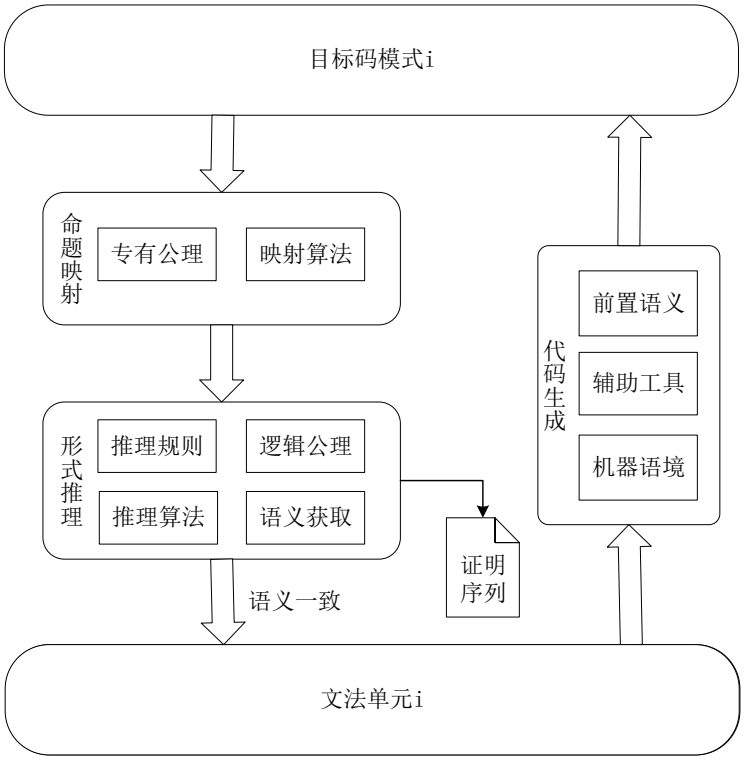

图 6　文法单元验证架构

图6中，文法单元的验证架构主要包括代码生成、命题映射和形式推理三个部分。代码生成是根据识别出的文法单元生成对应的目标码模式，这一过程涉及到了目标机器的语境和代码生成辅助工具，还需要提供文法单元的语义作为前置条件以进行后续的语义一致性确认。命题映射部分是使用专用公理把目标码模式映射为对应的目标码模式命题。其中，目标码模式在本文中特指的是 Power PC 汇编代码片段，专用公理是指前文所述的 Power PC 指令构成的指称语义集。形式推理部分是以目标码模式命题为前使用编译验证公理系统的推理规则和公理对其进行推理，把推理的结果进行语义获取操作即可得到目标码模式的语义。确认目标码模式的语义和作为前置条件的文法单元的语义是否一致并输出证明序列。

3.3.2 编译验证公理系统

编译验证公理系统由命题的符号定义、公理集和推理规则等组成，其本质是基于命题逻辑公理系统的，下面将从上述几个方面来介绍。

命题是具有确定真或假含义的陈述句，最简单的命题是原子命题。复合命题的新命题是由已知的原子命题用逻辑联结词组合而成的。在编译验证公理系统中，命题是由以下表 2 中的符号组成的公式。

表 2　公理系统的符号定义

类型	符号	意义
逻辑联结词	¬	一元联结词，非
	->	二元联结词，蕴含
	∨	二元联结词，或
	∧	二元联结词，且
寄存器元件	GPR	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	表示条件寄存器，用来反映某些操作的结果（比如比较指令），协助测试和分支转移指令的执行
	PC	表示程序计数器，用于存放下一条指令所在单元的地址的地方
内存	MEM	表示内存地址，与内存寻址相关
常量	7	表示十进制数 4
	b100	表示二进制数 100
	@	表示取地址操作
	.L1	表示相对地址
语句	<LOG-EXP>	表示逻辑表达式
	<ASS-EXP>	表示赋值表达式
	<STA-LIST>	表示复合表达式
运算符	+	二元运算符，加
	~	一元运算符，取非
	=	一元运算符，赋值
	<	二元运算符，小于

	>	二元运算符，大于
	==	二元运算符，等于
	||	二元运算符，或

公理是某些永真的命题。定理是指任何可证的命题，因此，出现在证明过程中的任何命题都是定理。本系统的公理集由逻辑公理和专用公理组成。其中，专用公理由 3.3.3 节中 Power PC 汇编指令的指称语义构成。逻辑公理如下：

公理 3.1 编译验证公理系统包含如下逻辑公理：

（i）$(\alpha \to (\beta \to \alpha))$

（ii）$((\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)))$

（iii）$(\neg\beta \to \neg\alpha) \to ((\neg\beta \to \alpha) \to \beta)$

这里α，β和γ可以是任意命题。

推理规则是正确推理的依据，任何一条永真蕴含式都可以作为一条推理规则。下表 3 给出本系统的推理规则，其中，*HYP* 为前提或者断言，P、Q为命题。

表 3　编译验证公理系统推理规则

名称	前件	后件
Rule1 CI 规则（合取引入）	$\begin{cases} HYP \vdash P \\ HYP \vdash Q \end{cases}$	$HYP \vdash P \wedge Q$
Rule2 合取删除规则	$HYP \vdash P \wedge Q$	$\begin{cases} HYP \vdash P \\ HYP \vdash Q \end{cases}$
Rule3 演绎规则	$HYP, P \vdash Q$	$HYP \vdash P \to Q$
Rule4 MP 规则（假言推理）	$\begin{cases} HYP \vdash P \\ HYP \vdash P \to Q \end{cases}$	$HYP \vdash Q$
Rule5 P 规则（前提引入）	在推导的任何步骤上，都可以引入前提。	
Rule6 T 规则（结论引用）	在推导任何步骤上所得结论都可以作为后继证明的定理。	

根据上述基本概念，下面给出编译验证证明系统的形式化定义。其中，在此系统下$\vdash_H$表示可证性。

定义 3.1 令Σ为一个命题集合。

（i）一个**从Σ出发的证明**（proof from Σ）是指一个由$\alpha_1,\alpha_2,...,\alpha_n$构成的有穷序列，使得对每个$i \le n$，以下三条之一成立：

（1）α_i是Σ中的元素；

（2）α_i是公理；

（3）α_i可以通过应用推理规则，由之前的某些α_j推出。

（ii）α_i是从Σ**出发可证的**（provable from Σ），记作$\Sigma \vdash_H \alpha$，如果存在一个从Σ出发的证明$\alpha_1,\alpha_2,...,\alpha_n$，这里$\alpha_n = \alpha$。

本文提出的形式化验证方法是基于上述系统的公理和推理规则的。对由目标码模式映射成的命题集选用上述公理和推理规则进行推理证明得到新命题，把新命题作为定理加入到目标码模式命题集中，重复上述过程直到目标码模式命题集遍历完成。把最终得到的命题集经过语义获取操作后得到每个目标码模式的语义，也就是文法单元的逻辑命题。可以看到，推导出的结果符合了按照 DO-178C 标准规定的编译验证系统开发与验证过程的高层需求。

3.3.3 编译验证形式语义

语义涉及程序文法结构上正确程序的含义，它研究语言与其所指对象间的关系。程序的语义是用一种元语言将程序加工数据的过程及其结果形式化来定义的。语义的形式描述是用严格的数学方法来研究程序，并对程序特性进行了精确定义，这十分有助于编译程序的设计。编译验证主要涉及到指称语义。

指称语义用与程序运行结果相对应的数学对象（即指称）来描述程序结构含义。指称语义仅关心程序的运行结果，不再关心程序如何执行。指称语义特有的、区别于其它语义形式的性质为：指称语义是可合成的，即任何表达式的含义由其子表达式的含义决定。指称语义较为常用，并且已经成了程序设计语言严格定义的基础。研究表明，可以自动把指称表示形式转换为等价的可以直接执行的形式。

在 32 位 Power PC 指令集范围内，本文根据 Power PC 官方文档[40]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 4 给出了部分简化后的 Power PC 汇编指令的指称语义。

表 4　Power PC 汇编指令的指称语义

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 4 中，GPR、CR、MEM 代表的含义如 3.3.2 小节中编译验证公理系统的符号定义所示。@target 表示相对地址，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。

最后，这些 Power PC 指令的指称语义由于已经在本课题组其他同学的工作中进行过验证，本文将把它们直接作为专用公理加入到编译过程的验证中。

3.3.4 编译验证核心方法

编译器本质上是一个符号转换程序，因此可以为编译过程建立完整的数学模型，利用这个模型可以对编译过程的正确性进行形式化验证。

设源代码为S，编译过程用$compile$表示，则由 2.3.1 小节可知编译过程正确性可以用如下等式来表示：

$$encode\left(semantic_{source}\left(S\right)\right) = semantics_{target}\left(compile\left(S\right)\right) \quad (1).$$

又令目标代码为T，目标代码是由源代码编译得到的，故有：

$$T = compile\left(S\right) \quad (2)$$

成立。由(1)、(2)式可以得到

$$encode\left(semantic_{source}\left(S\right)\right) = semantics_{target}\left(T\right) \quad (3).$$

不妨，再设 C 文法单元为S_i，对应的目标码模式为T_i。根据上下文无关文法的独立性，则有：

$$\begin{cases} S = \bigcup_{i=1}^{n} S_i \\ T = \bigcup_{i=1}^{n} T_i \end{cases} \quad (4).$$

把(4)式代入(3)式有：

$$encode\left(semantic_{source}\left(\bigcup_{i=1}^{n} S_i\right)\right) = semantics_{target}\left(\bigcup_{i=1}^{n} T_i\right) \Rightarrow$$

$$\bigcup_{i=1}^{n} encode\left(semantic_{source}\left(S_i\right)\right) = \bigcup_{i=1}^{n} semantics_{target}\left(T_i\right) \Rightarrow$$

$$encode\left(semantic_{source}\left(S_i\right)\right) = semantics_{target}\left(T_i\right) \quad \left(i = 1,...,n\right) \quad (5).$$

上述(5)式说明了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的等价性来实现。

编译验证方法的核心是对文法单元的验证。在安全 C 子集中，文法单元大体上可以分为三类，即表达式文法单元、条件选择结构文法单元和循环结构文法单元。对这三类文法单元的验证思想整体上是相同的，但具体实现中还是存在着一些差异的。依据文法单元是否包含循环结构，需要调用不同的算法分别进行循环结构文法单元和其它非循环

结构文法单元的验证。安全 C 子集中规定的其它文法成分，如 include 语句、宏定义、变量声明等，由于其结构过于简单，只要确认它们符合安全 C 语法定义即可保证自身的正确性，而这个过程可以在编译过程中完成，本文将不会对它们进行形式验证。图 7 给出了本文提出的形式验证方法的验证流程。

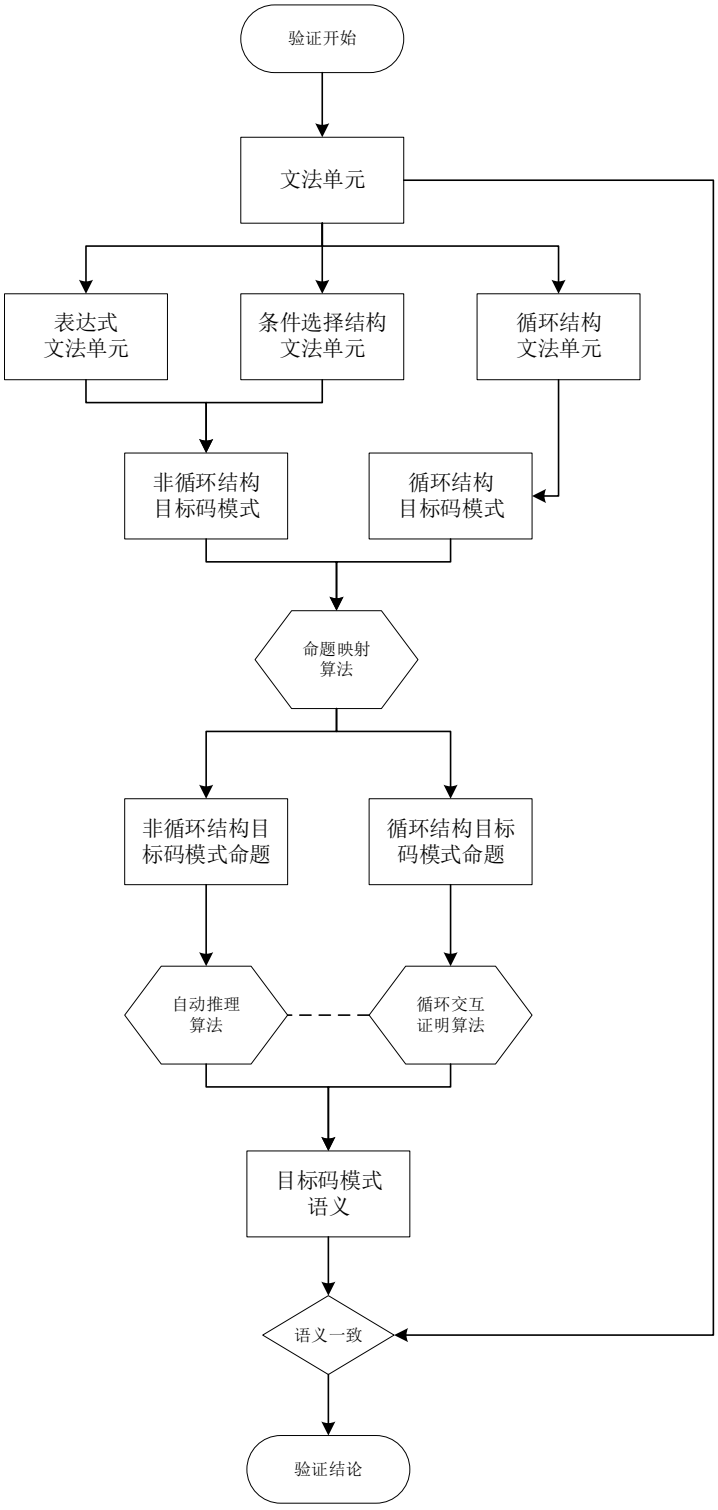

图 7　编译验证流程

下面给出编译形式化验证方法的核心点：

形式推理的正确性：证明过程中会依据当前命题的特点自动选择推理规则将命题转化为新的命题，并把新命题作为已证明的定理应用到后续的证明过程中。由于证明过程中每个步骤都是前提、公理或由推理规则推导出的定理，而命题逻辑公理系统是可靠且完全的，所以最终推导出来的证明结果一定是正确的。

命题映射算法：主要作用是基于专用公理将目标码模式转化为对应命题的表达形式，便于后续的形式推理。专用公理是由目标码指令集中每条指令的指称语义构成的。算法会遍历目标码模式的每条指令并用指令对应的指称语义来生成新命题，把得到的每个新命题加入到新命题集合中，从而获得整个目标码模式的命题集。命题映射算法在实际中被当做 Read 推理规则引入到了形式推理的过程中，从而简化了对整个验证过程的描述。

自动推理算法：本文形式验证方法的核心。算法从给定的前提—目标码模式命题集出发，利用本文公理系统的推理规则和公理对命题集进行形式推理。证明过程中需要对证明策略进行遍历搜索，并把得到的新命题作为定理加入到命题集中，依据每一步证明返回的结果判断是否需要去掉多余的旧命题，直到处理完整个命题集。

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。它主要用来对包含循环结构的目标码模式命题进行推理，整个处理过程类似于数学中的归纳法，但还需要外界提供循环不变式的帮助。在本文编译形式化验证方法中，循环不变式指的是文法单元的语义这个前置条件。

3.4 安全 C 编译器构建方法

3.4.1 安全 C 编译构建方法架构

本文中编译部分的构建主要分为前端和后端两个阶段。为了支持上文提出的形式验证方法，系统的前端必须支持文法单元处理等一系列操作。传统的编译实现技术中并不存在文法单元的概念，所以需要修改传统的编译技术或者完全自定义一套新的编译技术。鉴于自定义一套新的编译技术难度过大，本文采取的方式是保留传统前端实现中的词法分析部分，而在语法分析部分重新设计新的处理算法，使其支持对文法单元的处理。前端的输出为识别出的文法单元集，后端使用前端传递过来的结果生成对应的目标码模式，进行文法单元的形式验证后，集成为最终的目标代码。下面给出本文中编译验证系统的架构，如图 8 所示。

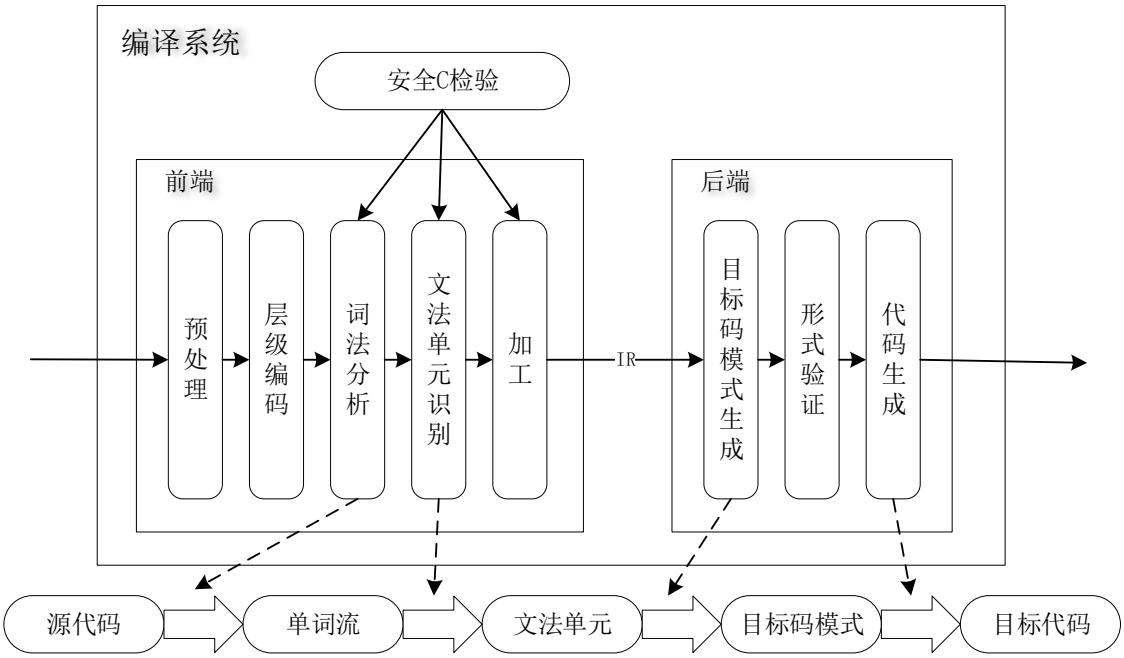

图 8　编译验证系统架构

图 8 中可分为上下两部分，上部分是编译验证系统的构成部分，下部分为输入的数据在整个处理流程中的转化过程。其中，上部分的构成又分为前端和后端两部分，前端部分结合实际的课题需求加入了层级编码、文法单元识别和安全 C 检验等过程，后端部分则相应的加入了目标码模式生成和形式验证等过程。形式验证是前文所述的文法单元的验证过程，本文将把它作为一个处理阶段加入到编译验证系统的后端实现中。

下面对编译系统构建方法还作出几点说明：

（1）预编译处理。预编译作为编译前期的工作，其主要的内容在于宏命令的开、正文替换和注释删除等。预编译器在工业上的标准实现中类似于编译器的实现内容，它也必须分析源代码的语义才能进行预处理过程。本文中简化了预编译处理的过程，把预编译阶段放到了编译器的前端实现中。

（2）编译遍历方式。编译器对源代码的编译遍历方式分为两种：一种是一遍编译，即把源代码只遍历一遍，在遍历的同时完成词法分析、语法分析等处理过程，虽然效率比较高，但实现起来非常复杂，不利于编译器代码的模块化设计；另一种是多遍编译，即在词法分析、语法分析、代码优化等每个阶段，都需要从头开始遍历源代码，虽然效率比较低，但获得源码的语义信息更为完整。本文中使用的也是多遍编译的方式，但不是每次都遍历源代码，而是遍历上一阶段的输出结果。

（3）高级语言规范。每种编译器都是对特定程序设计语言规范的实现。本文选择的程序设计语言规范是前文所述的安全 C 子集规范，它是 MISRA-C:2004 的子集且结合了

航天型号软件的特点。MISRA-C:2004 使用 C90 标准，即 ISO 9899:1990，定义安全关键领域的 C 编程规范。本文高级语言规范本质上是基于 C 程序设计语言的。

（4）汇编语言的处理。本文的编译验证系统产生的目标码指令属于 Power PC E500（PPC）处理器指令集。PPC 指令支持 32 位和 64 位两种模式，本文采用的是 32 位指令模式。编译器在产生汇编代码之前已经分析了源程序的正确性，生成的汇编代码都经过了形式验证，故生成的汇编代码都是合法的汇编指令。

（5）编译优化。编译器优化部分是现代编译器的重要组成结构，鉴于其不会影响源代码的语义和作者水平所限的缘故，本文讨论的编译系统构建技术将不包括编译优化这一过程。

3.4.2 安全 C 编译构建核心方法

编译器的前端将源程序的结构与程序设计语言规范进行比较，以对其进行检验。文法是描述一门程序设计语言的定义和实现其编译器的方法，多用 BNF 范式来描述。BNF 范式是一种格式上有点改变和缩短了的上下文无关文法，有效的简化了程序设计语言的定义和编译程序的结构。下面将给出部分安全 C 子集的文法，如下表 5 所示，完整的文法见附录 I。

表 5　安全 C 子集文法

<IDENTIFIER> ::= <ALPHABET> <LETTER> | <ALPHABET>
<LETTER> ::= <ALPHABET> <LETTER> | <ALPHABET> | <DIGIT> <LETTER> | <DIGIT> | _<LETTER> | _
<ALPHABET> ::= a | b | c | ..... | z | A | B | C | ..... | Z
<DIGIT> :: = <ZDIGIT> | <NZDIGIT>
<ZDIGIT> ::= 0
<NZDIGIT> :: = 1 | 2 | ..... | 9
<CONSTANT> ::= <INTNUM> | <DECIMALNUM>
<INTNUM> ::= <DIGIT> <INTNUM> | <DIGIT>
<DECIMALNUM> ::= <INTNUM> . <INTNUM>
<statement-list> ::= <statement> <statement-list> | <statement>
<statement> ::= <if-statement> | <switch-statement> | <while-statement> | <do-statement> | <for-statement> | <jump-statement> | <empty-statement> | <assignment-statement>

```
<if-statement> ::= if '( <logical-expression> ') <statement> |
                   if '( <logical-expression> ') <statement> else <statement> |
                   if '( <logical-expression> ') '{ <statement-list> '} |
                   if '( <logical-expression> ') '{ <statement-list> '} else '{ <statement-list> '}
<while-statement> ::= while '( <logical-expression> ') '{ <statement-list> '}
<do-statement> ::= do '{ <statement-list> '} while '( <logical-expression> ')
<for-statement> ::= for '( <for-assignment-expression> ; <logical-expression> ;
<for-assignment-expression> ') '{ <statement-list> '}
<for-assignment-expression> ::= <variable> = <int-expression>
...
```

表 5 中，文法的描述被分为了两个部分，其中上部分描述的是文法中的词法规则，下部分描述的是文法中的语法规则，词法规则是语法规则的构成基础。文法的词法规则由词法分析部分来实现，主要的任务是将字符流转化为特定输入语言的单词流。文法的每一条语法规则都对应着一个 C 文法单元。文法单元是文法在特定程序语言的一般化表示，它的识别处理需要自定义算法来实现。下面我们将讨论一下前端中这两部分所涉及的技术。

词法分析中主要涉及到的技术是有限自动机（FA）。有限自动机是一个五元组 $(S,\sum,\delta,s_0,S_A)$，其中个各分量的含义如下：

（1）S 是有限状态集合，包括一个错误状态 S_e。

（2）$\sum$ 是一个有限字母表。

（3） $\delta(s,c)$ 是一种状态转移函数。它将每个状态 $s\in S$ 和每个字符 $c\in\Sigma$ 的组合 (s,c) 映射到下一个状态。在状态 s_i 遇到输入字符 c，FA 将采取转移 $s_i \xrightarrow{c} \delta(s_i,c)$。

（4）$s_0\in S$ 是指定的起始状态。

（5）S_A 是接受状态，$S_A\subseteq S$。

基于上述有限自动机，词法分析中识别每一个单词的过程为：$\delta(s_0,\mathrm{x}_1)$ 表示 FA 从起始状态 s_0 处理输入字符 x_1 所发生的状态转移。$\delta(s_0,\mathrm{x}_1)$ 产生的状态接下来作为输入，

该状态连同 x_2 又输入到 δ 产生下一个状态；依次类推，直至耗尽所有的输入。最后一次应用 δ 的结果仍然是一个状态。如果该状态是 S_A，那么串 $x_1x_2x_3...x_n$ 就被 FA 接受了；否则，FA 发现一个词法错误。FA 可能在某个状态处理字符 x_j 转移到错误状态 S_e，此时也是发生了词法错误。

文法单元识别中主要使用的技术是下推自动机。下推自动机是一个六元组 $(Q,\Sigma,\Delta,s_0,z_0,F)$，其中：

（1）Q 是状态的一个非空有穷集合。

（2）Σ 是输入字母表。

（3）Δ 是栈字母表。

（4）s_0 是起始状态。

（5）z_0 是栈起始符号。

（6）F 是终止状态集，$F \subseteq Q$。

（7）$\delta(s,\mathrm{c},\mathrm{z})=\left([Js,Rs],z'\right)$ 是状态映射函数。它表示从当前状态 s 出发，且栈顶符号是 z，若输入的字符为 c，则映射到三元偶 $\left([Js,Rs],z'\right)$。其中，Js 为跳转到的下一个状态集合，Rs 为跳转状态执行结束后返回的状态，z' 为动作结束后的栈顶元素。

基于上述下推自动机，文法单元的识别过程为：$\delta(s_0,\mathrm{x}_1,z_0)$ 表示下推自动机从起始状态 s_0 出发且单词栈顶元素为 z_0，处理输入单词 x_1 所发生的状态转移，用一个三元组 $\left([Js_0,Rs_0],z_0'\right)$ 表示，此时把 x_1 入单词栈顶，Js_0 压入状态栈顶。然后，从当前状态栈取出栈顶状态集合 Js_0，取出顶部的元素 Js_{0i} 作为当前状态进行 $\delta(Js_{0i},\mathrm{x}_2,\mathrm{x}_1)$ 的转移，把 x_2 入单词栈顶，Js_0 压入状态栈顶，不断重复上述转移过程。若到达文法单元对应的 FA 的终结状态，就完成了一个文法单元的识别，把 FA 最终的状态写入 Rs_0 返回。反之，若在转移过程中发生了错误，则需要进行状态的回溯操作，此时弹出单词栈顶的元素，弹出状态栈的栈顶状态集合 Js_0，取 Js_0 的顶部的元素 Js_{0i+1} 作为当前状态进行

$\delta\left(Js_{0i+1}, \mathrm{x}_2, \mathrm{x}_1\right)$的转移。

最后，当上述转移过程处理完所有的输入单词时，如果最终状态是 FA，那么就完成了源代码的文法单元识别过程；否则，源代码中出现了错误。

3.4.3 层级编码和安全 C 检验

层级编码是一种把源代码按照代码的嵌套层次依次赋予对应的数字编号的方法。层级编码方法是在编译的前端实现的，只用到了代码的少量语法信息。源代码层级编码的结果需要映射到编译出的目标代码及相应的证明序列中，以实现从源代码到目标代码及证明序列之间的相互追踪，从而完成 DO178C 标准中规定的 A 级软件可追踪性需求。实际中，源代码的层级编码结果还需要写回到新的代码文件中。下面给出源代码模板的层级编码示例，如表 6 所示。

表 6　层级编码示例

<header-file-list>	// 1
<macro-definition-list>	// 2
<type-specifier> main (<parameter-type-list>) {	// 3
<declaration-list>	// 3.1
…	
for (i = 1; i <= n; i++) {	// 3.7
tmp = i % 2;	// 3.7.1
if (tmp == 0) {	// 3.7.2
sum = sum + i;	// 3.7.2.1
} else {	// 3.7.3
sum = sum – i * 2;	// 3.7.3.1
}	// 3.7.4
}	// 3.8
…	
<return-statement>	// 3.18
}	// 4
<function-list>	// 5

表 6 中，源代码模板是安全 C 源代码的一般化表示形式，对其进行层级编码具一定的通用性，可以较好的表示对一般的安全 C 源代码的层级编码过程。从表中可以看出源代码模板的最外层编号为 1~5，进入主函数后会在当前编号的后面增加一个点号和层次数，其代表着第一层嵌套。退出主函数后，需要去掉当前编号的最右点号和点号后的层次数，并把得到的编号加 1 作为当前行的编号。依此类似，其它复合语句的编码过程和主函数相同。上述过程完成后，即可得到安全 C 源代码的层级编码结果。实际中，一般

使用栈的方式存储当前的层级编号，用入栈和出栈来对应复合语句的进入和退出过程。

安全 C 子集规定了本文编译验证系统处理的高级语言规范，任何不符合安全 C 子集规范的源代码都是不符合安全关键系统要求的。安全 C 检验是为了分析源代码是否符合安全 C 子集规范，它是实现系统高安全性和可靠性的重要保证。安全 C 检验可以分为直接检验和间接检验两种，具体架构如图 9 所示。

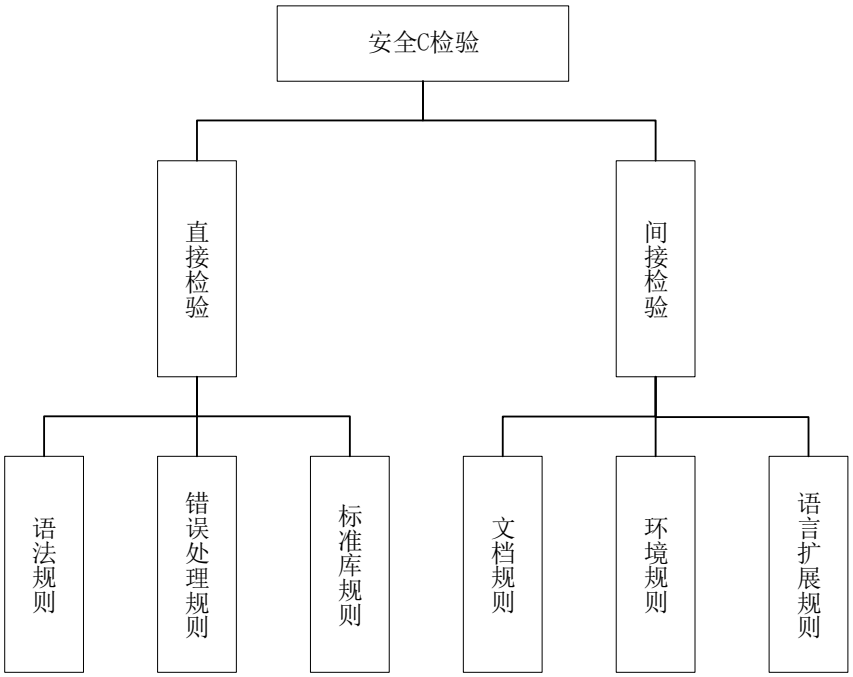

图 9　安全 C 检验架构

图 9 中，安全 C 直接检验是指在编译系统的构建中实现检验过程，其可以通过在编译前端的词法分析、文法单元识别等阶段增加检验函数或者设计检验算法来完成。直接检验包括安全 C 子集的语法规则、错误处理规则和标准库规则的检验过程，它是整个安全 C 检验的主要构成部分。间接检验是除去直接检验之外的其它过程，一般不能再编译构建中完成。间接检验包括文档规则、环境规则和语言扩展规则的检验过程。其中，文档规则规定了编写与源程序相关的文档需要遵循的标准。环境规则是对源代码所需遵循的其它规范的补充说明，如 ISO 9899:1990。语言扩展规则规定了在源代码中使用其它语言需要注意的地方，如汇编语言应该被封装并隔离等。

3.5 小结

本章主要论述了编译形式化验证方法和安全 C 编译器关键构建方法的设计，以及验证方法和构建方法的原理与架构，具体如下：

（1）提出了基于 DO-178C 规范的编译验证系统的开发与验证过程。系统需求是安

全关键编译规范，高层需求为文法单元的逻辑命题，低层需求为编译验证结构模型，软件架构是自动机的并集，软件编码与集成是把自动机的实现代码和编译验证代码集成为系统。

（2）提出了基于文法单元和目标码模式的编译语义验证方法，把编译过程正确性验证等价为对源代码中包含的每个 C 文法单元的验证。采用形式推理技术，基于本文构建的公理系统验证了编译前后每个 C 文法单元和对应的目标代码模式的语义是否一致，并给出了实现验证方法的整体架构。

（3）提出了安全 C 编译器构建中的关键方法，并给出了系统架构。为了支持编译语义验证方法的实现，采用了有限自动机和下推自动机作为编译前端的主体实现算法，解决了词法分析和文法单元的识别的问题。使用层级编码和安全 C 检验方法实现了 DO178C 规定的 A 级编译验证系统的可追踪性和高安全性需求。

第四章 编译形式化验证与安全 C 编译器构建的关键技术

4.1 编译形式化验证关键技术

4.1.1 文法单元和语义

在形式语言理论中，上下文无关文法是一种重要的变换文法，被广泛应用于程序设计语言的定义中。上下文无关文法之间具有相互独立的特性，因此上下文无关文法的并集仍然是上下文无关文法。安全 C 子集也是由上下文无关文法来规定的，每种上下文无关文法描述的语句结构的 C 语言表达形式就是相应的 C 文法单元。C 文法单元可以用对应文法的下推自动机来识别，不同文法单元之间之间也是相互独立的。文法单元的独立性是本文验证方法的基础。

语义定义了用于判定特定模型中的语句真值的规则，也就是语句的含义以及这些含义之间的关系。为了获得每个 C 文法单元的语义，本文引入了程序语境的概念，根据程序的语境可以定义出的文法单元的语义。语境在计算机程序中表示语法单位表达某种特定意义时所依赖的各种环境和上下文因素，如局部变量、全局变量等。下表 7 给出了部分 C 文法单元和其对应的语义。

表 7　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 7 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。

4.1.2 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。目标代码是目标码模式在目标机器的具体语境下集成而得到的。编译正确性验证的目的是证明每个 C 文法单元的语义和对应的目标码模式的语义是否一致。下表 8 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 8　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>
	}
		<LOG-EXP>
		cmpi 7,0,0,0
		beq 7,.L1
		<STA-LIST_1>

	else
	{
	<STA-LIST_2>
	}
		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

目标码模式命题是目标码模式的命题化表示形式，可以基于 3.3.2 节中 Power PC 汇编指令的指称语义使用命题映射算法而得到。目标码模式只有表示成命题的形式才能进行后续的形式推理，一般把目标码模式命题集作为前提集，其包含的符号由 3.3.2 节中的编译验证公理系统的符号定义来限定。下表 9 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命题。

表 9　目标码模式命题

语句	目标码模式	目标码模式命题
<if_else-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)
<do-while-statement>	.L1: <STA-LIST> <LOG-EXP>	P1 = .L1: P2 = <STA-LIST> P3 = GPR[0] = <LOG-EXP>

	cmpi 7,0,0,0 bne 7,.L1	P4 = GPR[0] < 0 -> CR[7] = b100 ‖ GPR[0] > 0 -> CR[7] = b010 ‖ GPR[0] == 0 -> CR[7] = b001 P5 = CR[7] == b100 -> PC = PC + @.L1 ‖ CR[7] == b010 -> PC = PC + @.L1 ‖ CR[7] == b001 -> PC = PC + 4
<for-statement>	<ASS-EXP_1> b .L2 .L1: <STA-LIST> <ASS-EXP_2> .L2: <LOG-EXP> cmpi 7,0,0,0 bne 7,.L1	P1 = <ASS-EXP_1> P2 = PC = PC + @.L2 P3 = .L1: P4 = <STA-LIST> P5 = <ASS-EXP_2> P6 = .L2: P7 = GPR[0] = <LOG-EXP> P8 = GPR[0] < 0 -> CR[7] = b100 ‖ GPR[0] > 0 -> CR[7] = b010 ‖ GPR[0] == 0 -> CR[7] = b001 P9 = CR[7] == b100 -> PC = PC + @.L1 ‖ CR[7] == b010 -> PC = PC + @.L1 ‖ CR[7] == b001 -> PC = PC + 4

4.1.3 编译验证证明方法

本文提出的证明方法是基于命题逻辑构成的编译验证公理系统，从公理系统中事先给定的前提（目标码模式命题）出发，根据公理系统的推理规则推导出一系列新命题，并作为定理加入到之后的证明过程中。由于证明过程中每一项都是前提、公理或由定理推导得到，而命题逻辑公理系统是可靠且完全的，所以最终推导出来的证明结果一定是正确的。

证明序列是一系列证明步骤的集合，每个步骤包括公式和证据两项。公式是形式推理的命题变换过程和结果，证据表示构造这一步骤的原因。证据可分为前提、定理和推理规则三种。当引用前提或者定理时，把前提或者定理的命题公式写到公式栏，并在证据栏标上公式的序号；当使用推理规则时，直接将产生的新命题写到公式栏，并在证据

栏标上推理规则的简写和旧命题的序号。

目标码模式按照是否含有循环结构可分为两类，即非循环结构目标码模式和循环结构目标码模式。对于两者映射而成的目标码模式命题根据是否含有循环结构，形式证明过程也不尽相同，含有循环结构的目标码模式需要使用循环不变式进行证明，而非循环结构的目标码模式只需直接进行形式推理即可。我们以<if-statement>和<while-statement>文法单元对应的目标码模式作为含有循环结构和不含有循环结构的代表来描述完整的形式证明过程。下面将给出<if-statement>文法单元对应的目标码模式命题的证明过程，如表 10 所示。

表 10　<if-statement>证明过程

公式	证据
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 ∧	S5, S6, S7, S8, S9, S10, CI

.L1: ∧ <STA-LIST_2> ∧ .L2: }	

S12= {
(<LOG-EXP> < 0 -> <STA-LIST_1>) ‖ (<LOG-EXP> > 0 -> <STA-LIST_1>) ‖ (<LOG-EXP> == 0 -> <STA-LIST_2>)
}
	S11, REDUCE

S13= (<LOG-EXP> != 0 -> σ(<STA-LIST_1>) ‖ <LOG-EXP> == 0 -> σ(<STA-LIST_2>))	S12, σ

表 10 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 S13，即：

σ(<LOG-EXP>) -> σ(<STA-LIST_1>) ‖ ~σ(<LOG-EXP>) -> σ(<STA-LIST_2>)

结合 4.4.1 节中<if-statement>的语义可知，二者语义保持了一致性，证毕。下面将给出<while-statement>文法单元对应的目标码模式命题的证明过程，如表 11 所示。

表 11　<while-statement>证明过程

公式	证据
S1 = PC = PC + @.L2	P1
S2 = .L1:	P2
S3 = <STA-LIST>	P3
S4 = .L2:	P4
S5 = GPR[0] = <LOG-EXP>	P5
S6 = GPR[0] < 0 -> CR[7] = b100 ‖ GPR[0] > 0 -> CR[7] = b010 ‖ GPR[0] == 0 -> CR[7] = b001	P6
S7 = <LOG-EXP> < 0 -> CR[7] = b100 ‖ <LOG-EXP> > 0 -> CR[7] = b010 ‖ <LOG-EXP> == 0 -> CR[7] = b001	S5,S6,MP
S8 = CR[7] == b100 -> PC = PC + @.L1 ‖ CR[7] == b010 -> PC = PC + @.L1 ‖ CR[7] == b001 -> PC = PC + 4	P7
S9 = <LOG-EXP> < 0 -> PC = PC + @.L1 ‖ <LOG-EXP> > 0 -> PC = PC	S7,S8,MP

+ @.L1 ‖ <LOG-EXP> == 0 -> PC = PC + 4	
S10 = (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)	S1, S2, S3, S4, S9, CI
S11 = (<LOG-EXP> != 0 -> <STA-LIST> ‖ <LOG-EXP> == 0 -> null)	S10, REDUCE
S12 = (<LOG-EXP> != 0 -> σ(<STA-LIST>) ‖ <LOG-EXP> == 0 -> skip)	S11, σ

由表 11 可知，对于类似于<while-statement>含有循环结构的目标码模式命题进行直接推理后，得到的目标码模式无法保持源代码中循环结构的语义，这是由于命题逻辑系统证明循环结构的局限性所导致的。本文引入了限定数学归纳法对循环结构目的标码模式命题进行证明，可以直接验证 C 文法单元和对应的含有循环结构的目标码模式的语义是否一致。

一般数学归纳法的逻辑表达式为 $P(0)\wedge(\forall n)\big(P(n)\to P(s(n))\big)\to(\forall n)P(n)$，限定数学归纳法是在一般数学归纳法的基础上限制 n 是有穷的，即程序的循环结构是可终止的。下面将结合<while-statement>文法单元的语义，对表 11 中 S10 公式使用限定数学归纳法来证明，如下表 12 所示。

表 12　<while-statement>限定数学归纳法证明过程

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。

4.2 编译形式化验证关键算法

4.2.1 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。下面将给出命题映射算法的伪代码，如表 13 所示。

表 13　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**

算法首先需要把 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。然后，逐条遍历输入的目标码模式，把目标码模式中每一条目标码的指令名和参数分离，并去掉无关的符号。接着通过判断指令名和参数构成的列表长度来决定指令的类型，若列表长度为 0 时，则是空行，不进行后续的处理；若列表长度为 1 时，则可以直接把目标码加入命题集；若列表长度大于 1 时，则需要用指令对应的指称语义来替换目标码，再将新产生的命题加入命题集中。最后，完成上述目标码模式的遍历后，输出得到的命题集。

表 13 中，命题映射算法的主体为一个 for 循环，for 循环的次数与目标码模式的规模呈线性关系，for 循环的内部也不在含有其它的循环结构，所以算法的时间复杂复为 O(n)，效率非常高。注意这里假设了算法中其它函数的处理时间为一个常数。

4.2.2 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 4.2.1 中命题映射算法输出的命题集合作为输入，首先新建一个空的命题集合，然后遍历输入的命题集合。若新命题集合为空，则直接把当前遍历到的命题 p 加入到新命题集合中；否则，需要遍历新命题集合，再进行处理。把新命题集合中的每个命题 q 与当前遍历到的命题 p 使用本文公理系统的某条推理规则进行推导，若推导出的结果为 null，则说明两个命题没有直接的关系可忽略；反之，若产生了新的命题，则直接加入到新命题集中。在推理的过程中，算法还会对命题 q 的内容作出一定的修改，若 q 的命题内容被清空时，则说明命题 q 需要删除，因为新产生的命题已经包含有 q 的内容。最后，只需对新命题集使用语义获取操作即可得到目标码模式的语义作为输出。命题自动推理算法的伪代码如表 14 所示。

表 14　自动推理算法

Algorithm 2 Automatic Derivation
Input: PropositionSet
Output: SemantemeSet
1: **for** each p in PropositionSet **do**

```
2:      if newPropositionSet.size() = 0 then
3:          add p to newPropositionSet
4:      else
5:          for each q in newPropositionSet do
6:              newProposition ← applyDerivationRuleToTwoPropositions (p, q)
7:              if newProposition is not null then
8:                  add newProposition to newPropositionSet
9:              end if
10:             if q's content is empty then
11:                 remove q from newPropositionSet
12:             end if
13:         end for
14:     end if
15: end for
16: for each p in newPropositionSet do
17:     s ← obtainSemantemeFromProposition (p)
18:     add s to SemantemeSet
19: end for
```

表 14 中，自动推理算法的核心也是一个 for 循环，循环的次数与输入的命题集规模呈线性关系。在外部 for 循环之内还有一个遍历新产生命题集的 for 循环，新产生的命题集合最大的规模不会超过输入命题集合的规模，故设输入的命题集规模为 n 时，则算法的时间复杂度为 $O(n^2)$，效率有待提升。

4.2.3 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，构造新的命题加入到 4.2.1 小节命题映射算法输出的命题集的一个拷贝中，调用 4.2.2 中的自动推理算法对新拷贝的命题集进行推理，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 15 所示。

表 15　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
```

由表 15 可知，循环交互证明算法其实只需要执行两遍即可，第一遍是证明初始阶段 n 为 1 时，待证结论是否成立；第二遍假设第 n 为 N 时待证结论成立，推导 n 为 N+1

时待证结论是否成立。外部 for 循环只执行了两遍，故算法复杂度为常数。又由算法内部两次调用了自动推理算法，故最终算法的复杂度为 $O(n^2)$。

4.3 安全 C 编译器构建关键技术

4.3.1 安全 C 词法分析方法

词法分析是编译处理流程中重要的一步，它将输入的字符流变换为输入语言的单词流。词法记号是程序设计语言代码的基本单位，通常可以分为标识符、关键字、常量、界符四大类，可以认为程序设计语言是词法记号按照一定规则的组合。识别出不同形式的词法记号是词法分析最基本的任务，为了从输入的字符流中分析出每个词法记号，需要引入有限自动机。

每个有限自动机都能解析并识别出专门的词法记号，比如识别标识符的有限自动机、识别常量的有限自动机等。有限自动机从开始状态启动，读入一个字符作为输入，并根据该字符选择进入下一个状态。接着继续读入新的字符，直到遇到结束状态为止，读入的所有字符序列便是有限自动机识别的词法记号。有限自动机分为两类，即确定的有限自动机（DFA）和非确定的有限自动机（NFA）。由于每个 NFA 都可以转换为一个 DFA，且 DFA 的实现较为简单，故本文使用 DFA 来描述所有词法记号的定义。下面将给出用 DFA 来描述标识符和整数的识别过程，并基于二者进行一些更深入的讨论。

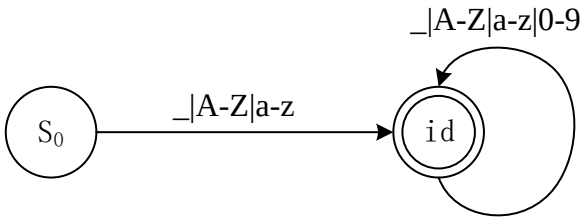

图 10　标识符有限自动机

图 10 中，标识符的识别从 S_0 状态开始，若读入的字符是下划线或者字母时，则进入状态 id。状态 id 是结束状态，其本身可以接收任意多个下划线、字母和数字，若此时读入其它字符时便停止自动机的识别过程。注意，此处省略了错误处理状态。这正好符合标志符在 C 语言中的定义：C 语言规定标识符只能由字母、数字和下划线 3 种字符组成，并且必须以字母或下划线开头。

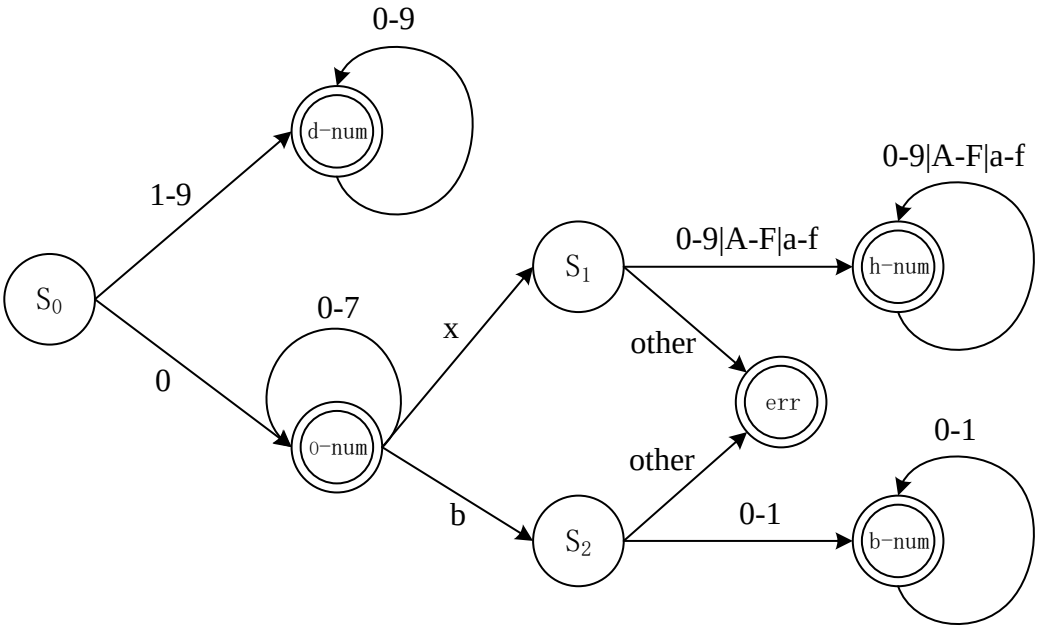

图 11　整数有限自动机

整数的有限自动机如上图 11 所示，其中结束状态 d-num 表示十进制整数，o-num 表示八进制整数，h-num 表示十六进制整数，b-num 表示二进制整数，err 表示自动机识别过程中出现词法错误，此时需要停止自动机。整数的识别是从 S_0 状态开始的，若读入字符是 1-9 时，进入 d-num 状态进行十进制整数的识别；若读入字符是 0 时，转移到 o-num 状态，然后继续读入字符，如果是 0-7，则正在识别八进制整数，如果读入的字符是 b，则进行二进制整数的识别，如果读入的字符是 x，则进行十六进制整数的识别。

其它词法记号的识别自动机与上述标识符和整数的自动机类似。下面给出一个基本的有限自动机识别算法作为所有词法符号自动机实现的抽象，如表 16 所示。

表 16　有限自动机识别算法

Algorithm 4 Finite Automata Recognition Algorithm
Input: CharStream, S, S_A **Output:** Token
1: char ← NextChar() 2: state ← S_0 3: **while** char ≠ *eof* and state ≠ S_e **do** 4:　　add char to token 5:　　state ← NextState(state, char) 6:　　char ← NextChar()

```
7: end while
8: if state ∈ S_A then
9:     report acceptance
10:    return token
11: else
12:    report failure
13: end if
```

上述算法中，S 为自动机的有限状态集合，S_A 为接受状态集合，S_e 代表错误状态，NextState 函数根据当前的状态和输入字符确定跳转的下一个状态，若输入字符非法，则下一个状态就为错误状态 S_e。NextChar 函数表示从当前输入流中读取下一个字符。循环结束后需要判断 state 的状态，若 state 的状态是 S_A 接受状态集合中的一个元素，则说明词法记号识别成功，返回识别出的 token；反之，若 state 状态为其它状态，则说明出现了词法错误，需要给出错误提示。

有了每个词法记号的自动机后，如何把多个自动机组合起来形成一个词法分析模块就是下一步需要解决的问题，这里使用到了自动机的并操作。下图 12 展示了把多个自动机并联为一个更复杂的自动机的方法。

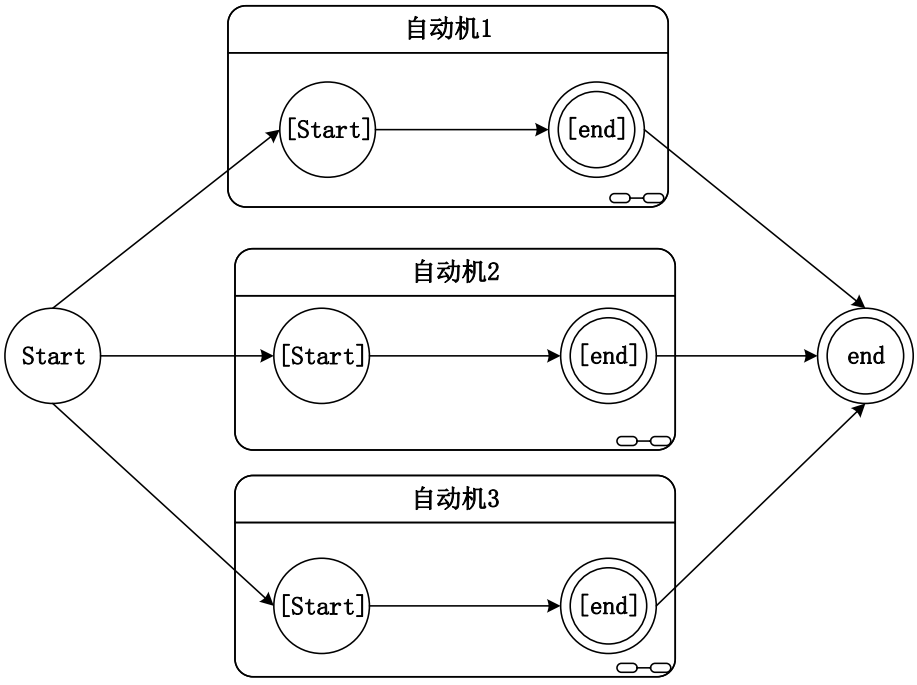

图 12　自动机的并操作

词法分析中在起始状态读入一个字符时就可以知道这个字符走的是哪一个分支的自

动机，因此只需要事先把所有分支的自动机构造出来，并把起始状态连接到所有分支的起始状态上，结束状态连接到所有分支的结束状态上即可。图 12 正是这一方法的体现，中间的自动机可以为前文所述的识别标识符自动机和整数自动机，此时从所有分支的起始状态转到标识符自动机的条件就是当前字符为为下划线或字母，而转到整数自动机的条件是当前字符为数字，自动机结束后可以自动转到所有分支的结束状态上，这样就完成了词法记号的所有识别过程。

4.3.2 文法单元识别方法

文法单元识别是一个从输入的单词流中构造出文法单元的过程，这各过程类似于传统编译技术中的语法分析。文法单元识别的目的是将多个具有语法范畴的单词匹配到程序设计语言的语法模型从而得到具有一定语义的语句单元。4.1.1 小节给出了安全 C 子集中的部分文法单元，仔细分析不难发现文法单元与词法符号的识别过程具有许多相似性。词法分析规定了如何将字符组合为单词，文法单元识别则定义了如何将单词组合成文法单元，二者可以使用相同的技术来处理。我们也使用了有限自动机来进行文法单元的识别，下面首先给出<if-statement>文法单元，如下表 17 所示。

表 17　<if-statement>文法单元

```
if (<LOG-EXP>)
{
    <STA-LIST_1>
}
else
{
    <STA-LIST_2>
}
```

构造识别<if-statement>文法单元的有限自动机的方法和词法分析中一致，但由于文法单元的识别自动机输入的是一个一个的单词，所以需要对自动机作出一定的改变，即把原始自动机中输入的字符替换为单词，把状态转移函数中的字符参数转变为单词参数，原始的有限字母表转换为有限单词表等。下面给出识别<if-statement>文法单元的有限自动机，如图 13 所示。

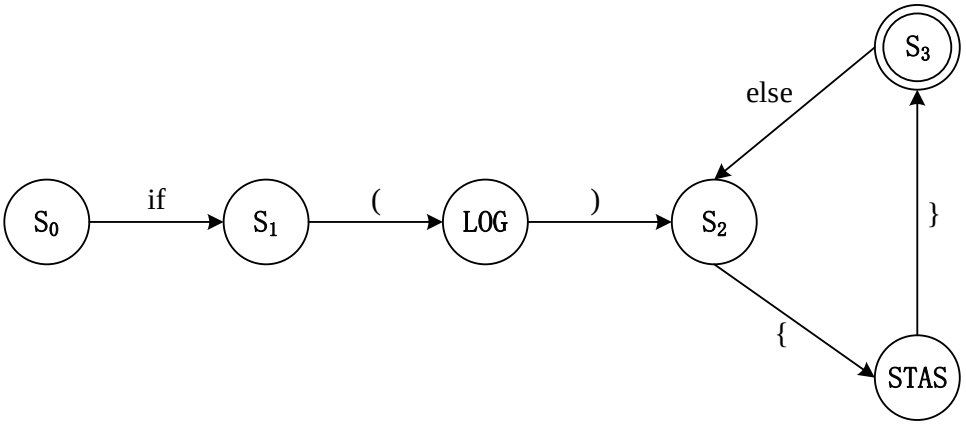

图 13　<if-statement>文法单元有限自动机

图 13 中，两个状态之间转移的条件是单词，故“(”、“)”、“{”和“}”均不是代表字符而是词法符号。LOG 是<LOG-EXP>的简写，它与一个识别逻辑表达式文法单元的子自动机相连。STAS 是<STA-LIST>的简写，它与一个识别复合表达式文法单元的子自动机相连。<if-statement>文法单元的识别是从 S_0 开始的，若遇到“if”单词时，就进入 S_1 状态。然后，继续读入单词，若遇到单词“(”时，则进入 LOG 状态直到 LOG 对应的子自动机识别完成，再次读入单词。若读入的是“)”，则进入 S_2 状态，继续读入单词。若遇到“{”，则进入 STAS 状态直到 STAS 对应的子自动机识别完成，再次读入单词。若读入的单词为“}”，则进入 S_3 结束状态，此时若继续读入的单词不是“else”，则此次文法单元的识别过程完成，否则转到 S_2 状态继续 else 复合表达式文法单元的识别。

安全 C 子集中其它文法单元的识别过程与<if-statement>文法单元相同，都可以建立相应的有限自动机来识别，自动机的实现算法同 4.3.1 节的有限自动机识别算法。如何把每个文法单元对应的自动机组合起来形成一个文法单元识别模块，这是下一步需要考虑的问题。在词法分析中只需前瞻一个字符就可以知道走哪一个分支的自动机，但文法单元的识别需要前瞻的单词数量却是不定的，如函数声明文法单元的识别需要前瞻 3 个单词才能确定，即“type-specifier function-name (”中除了前两个为类型和标识符外，第三个单词必须是“(”；否则，若只关注前两个单词，那么可能进入变量声明文法单元的识别分支。这种现象就是识别过程的二义性，所以不能直接使用并操作来组合识别每个文法单元的自动机。

在<if-statement>文法单元有限自动机中，还包含了 LOG 和 STAS 两个复合状态，在其对应的子自动机执行完毕后会返回父自动机的当前状态，所以在进入子自动机之前需要一个栈来保存父自动机的现场，这也是有限自动机无法实现的。基于上述两个原因，

下面给出下推自动机算法作为所有文法单元自动机的组合方法，如表 18 所示。

表 18　下推自动机算法

Algorithm 5 Pushdown Automata Algorithm

Input: TokenStream, Q, F

Output: Collections<SyntaxUnit>

```
1: build AnalysStack<Token> and RollbackStack<Stack>
2: while TokenStream not eof do
3:     token ← NextToken ()
4:     state ← S0
5:     statesStack ← NextState (state, token, ε)
6:     push token to AnalysStack
7:     push statesStack to RollbackStack
8:     while RollbackStack not empty do
9:         pop the top of RollbackStack to statesStack
10:        while statesStack not empty do
11:            pop the top of statesStatck to state
12:            if state ∈ F then
13:                construct SyntaxUnit from AnalysStack
14:                add SyntaxUnit to Collections
15:                clear AnalysStack, statesStatck and RollbackStack
16:            end if
17:            if state is compound_state then
18:                (Rs, z') ← SubAutomata (state, TokenStream)
19:                if Rs ≠ Se then
20:                    push z' to AnalysStack
21:                else
22:                    ResetTokenStream ()
23:                    continue
24:                end if
25:            end if
```

```
26:                token ← NextToken ()
27:                tmpStatesStack ← NextState (state, token, top of AnalysStack)
28:                push token to AnalysStack
29:                push statesStack to RollbackStack
30:                push tmpStatesStack to RollbackStack
31:                break
32:            end while
33:            if statesStack is empty then
34:                pop the top of AnalysStack
35:            end if
36:        end while
37: end while
```

上述算法中，需要定义一个分析栈、一个回滚栈以及一个状态栈。分析栈中存储当前自动机已经匹配的单词；回滚栈中记录了每个阶段自动机都可能到达的状态集合，当自动机匹配失败时可以回溯到上一个状态，从而可以继续当前的匹配过程；状态栈中存储当前阶段自动机可能达到的状态。

下推自动机算法以新建一个分析栈和回溯栈来开始，主体为一个遍历输入单词流的循环。算法首先读入一个单词并把当前状态设置为 S_0，使用 NextState 状态映射函数获得下一阶段的状态集合，注意 NextState 在当前参数下可能返回空状态集合。然后，把当前读入的单词放入分析栈，下一个阶段的状态集合放入回滚栈。然后，使用一个循环来遍历回滚栈，从回滚栈顶取出一个状态集合作为当前状态集合，然后遍历当前状态集合。若当前状态为终止状态集合 F 中的一个元素，则完成了一个自动机的识别过程，把分析栈中的单词构造成文法单元并加入到输出的文法单元集合中；若当前遍历到的状态为复合状态，则进入子自动机的识别过程，子自动机会返回其到达的状态 Rs 和文法单元的标识符，把标识符加入分析栈。若 Rs 为错误的状态，则子自动机识别过程出错，需要回溯输入单词流，这个过程由 ResetTokenStream 函数完成的。当遍历完当前状态集合却没有一个状态能最终到达自动机的终止状态，则说明父状态无效且在此状态输入的单词也是无效的，需要从分析栈中取出栈顶的元素并从回滚栈顶取出下一个状态集合作为当前状态集合。最后，不断重复上述过程，直到输入的单词流 TokenStream 遍历完成或回滚栈为空而止。

4.3.3 安全 C 层级编码方法

层级编码方法是构造符合 A 级软件标准编译验证系统的重要保证。实践中输入的源文件一般包含多个子文件，不同的子文件是通过 C 语言内置关键字来互相引用，故需要一定的语法信息才可以找到这些文件并合理的组织它们。下面将给出基于词法分析的层级编码算法，如表 19 所示。

表 19　层级编码算法

Algorithm 6 Hierarchical Coding Algorithm
Input: SourceCode **Output:** TokenStream
1: build stack, libsList 2: push 1 to statck 3: **for** each line in SourceCode **do** 4:　　**if** line contains '}' **then** 5:　　remove the top of stack 6:　　pop the top of stack to tmp 7:　　push (tmp + 1) to stack 8:　　**end if** 9:　　**if** line.length ≠ 0 **then** 10:　　tmpLibs ← solveLine (line, stack, TokenStream) 11:　　add tmpLibs to libsList 12:　　**end if** 13:　　**if** line contains '{' **then** 14:　　push 0 to statck 15:　　**end if** 16:　　**if** line.length ≠ 0 **then** 17:　　pop the top of stack to tmp 18:　　push (tmp + 1) to stack 19:　　**end if** 20: **end for** 21: **for** each lib in libsList **do**

22: solveMultipleFile (lib, stack, TokenStream)

23: **end for**

上述算法，以新建一个编号栈和文件名列表开始。首先，向编号栈中放入元素 1，并以行为单位遍历输入的源代码，若当前行中含有‘}’符号，则说明此行为复合语句的出口，需要弹出编号栈栈顶元素并使层级减一，同时把弹出元素后的编号栈栈顶元素加一。然后，若当前行长度不为 0，则交给 solveLine 函数来识别当前行中的单词，返回当前行中可能存在的代表文件名的标识符并加入文件名列表中。若当前行含有‘{’符号，则说明此行为复合语句的入口，需要向编号栈放入 0 元素。然后，若当前行长度不为 0，则使编号栈栈顶元素加一。最后，遍历所得到的文件名列表并把每个引用的文件交给 solveMultipleFile 函数来处理，solveMultipleFile 函数会递归调用层级编码算法来对每个文件分别进行处理。

4.4 小结

本章主要针对编译形式化验证与安全 C 编译系统构建过程中所用到的关键技术进行研究，具体如下：

（1）定义了编译形式化验证过程中的关键概念，即文法单元和语义、目标码模式和命题。基于上述概念使用形式推理的方法证明了文法单元和目标码模式语义的一致性，并且对于含有循环结构的文法单元使用了限定数学归纳法来证明，最后给出了整个证明过程的证明序列。

（2）提出了三个应用于编译形式化验证过程中的算法。命题映射算法的作用是把目标码模式转换为命题的形式，便于后续的推理和证明；自动推理算法是验证过程的核心，它基于本文构建公理系统，使用前提集和推理规则推导出一系列新命题，把这些新的命题作为定理加入前提中进行后续的证明；循环交互证明算法主要用于含有循环结构的文法单元的证明，其理论基础是限定数学归纳法。

（3）提出了构建安全 C 编译系统的关键方法。使用有限自动机识别算法和自动机的并操作完成了源代码中词法记号的识别，设计了下推自动机算法从单词流中识别出了不同的文法单元。最后，提出了层级编码的方法实现了 DO178C 规定的 A 级软件可追踪性需求。

第五章 编译验证工具的设计与实现

5.1 编译验证工具架构

本文设计实现的编译验证工具主要工作流程为：对输入的源代码进行预处理和词法分析把输入的字符流转化为单词流；使用下推自动机算法把单词流转化为文法单元集合；生成文法单元对应的目标码模式并进行文法单元和目标码模式的语义一致性验证；最后，进行目标代码的集成和生成。基于上述工作流程，下面将给出编译验证工具的系统框架，如图 14 所示。

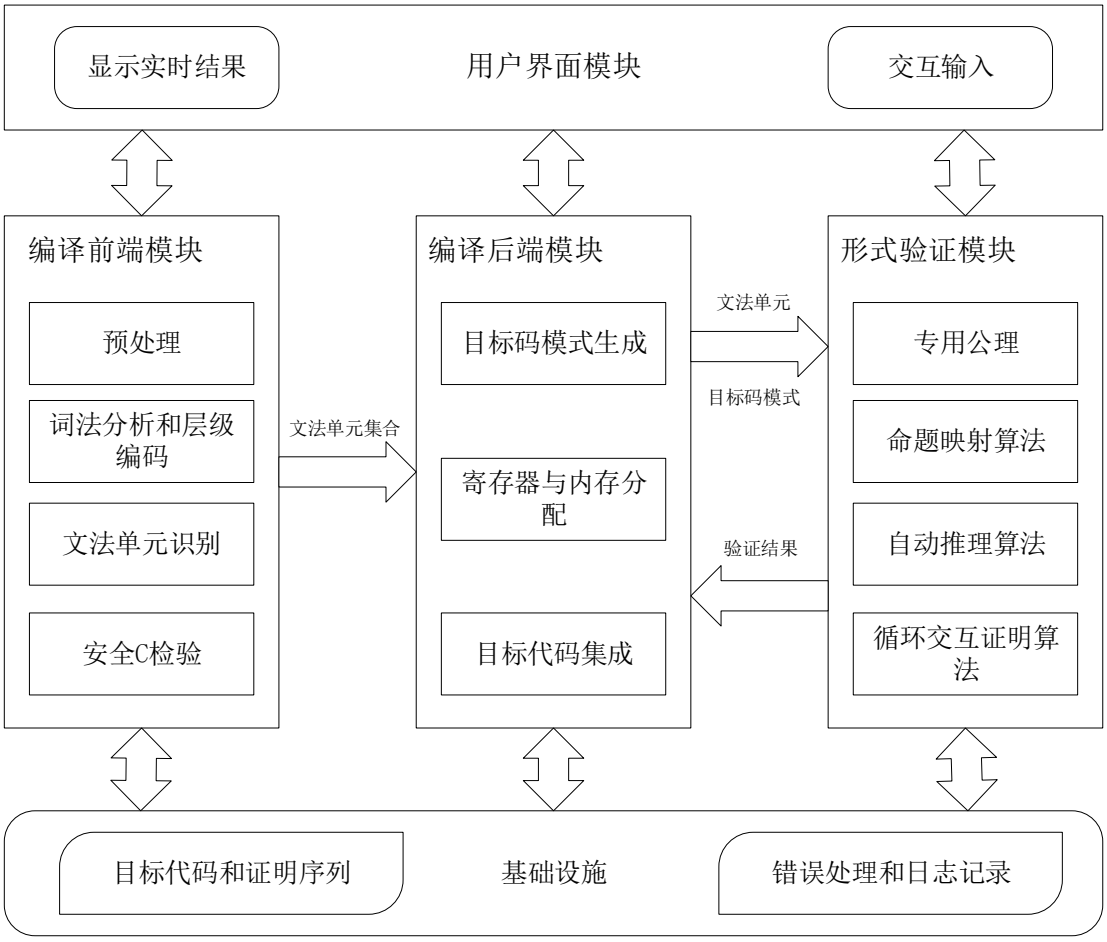

图 14　编译验证系统框架

图 14 中，编译验证系统主要由四个模块组成，分别为编译前端模块、编译后端模块、形式验证模块和用户界面模块，下面分别描述各个模块的功能。

编译前端模块：对用户输入的源代码进行处理以生成文法单元集合并传递给编译后端模块使用。主要的处理过程为首先预处理源程序去除掉无用的信息等，然后对其进行词法分析和层级编码把字符流转换为具有一定次序的单词流，接着使用下推自动机算法

从单词流中识别出文法单元并保存在集合中，最后需要进行安全 C 检验以保证输入的源代码符合安全 C 子集规范，检验过程是嵌套在前面的几个步骤中的。

编译后端模块：接收输入的文法单元集合并基于形式验证模块返回的结果决定是否生成目标代码。主要的处理过程为遍历输入的文法单元集合，对每个遍历到的文法单元生成相应的目标码模式，把文法单元和对应的目标码模式传递形式验证模块来验证。若文法单元的语义一致性验证通过，则依据当前的编译后端语境（寄存器和内存等），生成目标代码；否则，给出编译验证出错的相关信息。

形式验证模块：进行文法单元和目标码模式的编译语义形式验证。主要的处理过程为根据目标机器的专用公理使用命题映射算法把输入的目标码模式转换为命题的形式，对于不包含循环结构的目标码模式命题直接使用自动推理算法依据本文公理系统的推理规则和前提即可推导出其语义并进行语义一致性检验；但对于包含循环结构的目标码模式命题，则需要使用循环交互证明算法和循环不变式才能进行语义一致性的证明。

用户界面模块：用户与程序进行交互的接口，辅助编译和验证整个过程。用户使用界面工具向程序发送请求信息，程序接收用户请求消息后调用后台模块完成请求，返回处理结果并显示出来。

最后，图 14 中基础设施指的是一些基础的公共操作，与具体的业务模块进行了解耦。主要包括日志记录、错误处理以及目标码、证明序列和其它一些中间结果的生成和保存操作，其在整个程序中应用的非常广泛且比较简单，后文将不会单独赘述。

5.2 编译验证工具实现

5.2.1 编译前端模块

编译前端模块主要任务是理解源代码并将其分析结果记录下来，以供后端稍后使用。该模块主体为两个核心类 Lexer 和 Recognizer，其中 Lexer 完成对输入源代码的预处理、词法分析、层级编码和安全 C 检验等功能，Recognizer 主要任务是对文法单元进行识别和加工等。LexerUtils 和 RecognizerUtils 是两个辅助核心类工作的工具类，它们是从核心类中把一些重复使用的方法抽取出来所构成的，这些方法一般是静态方法。Token 类是词法分析所产生的结果，表示一个一个的单词。SyntaxUnitNode 类是构成文法单元的节点。SyntaxUnitCollections 类代表文法单元的集合，是前端经过一系列处理所生成的最终结果，也是后端模块的输入。下面将给出编译前端模块类图，如图 15 所示。

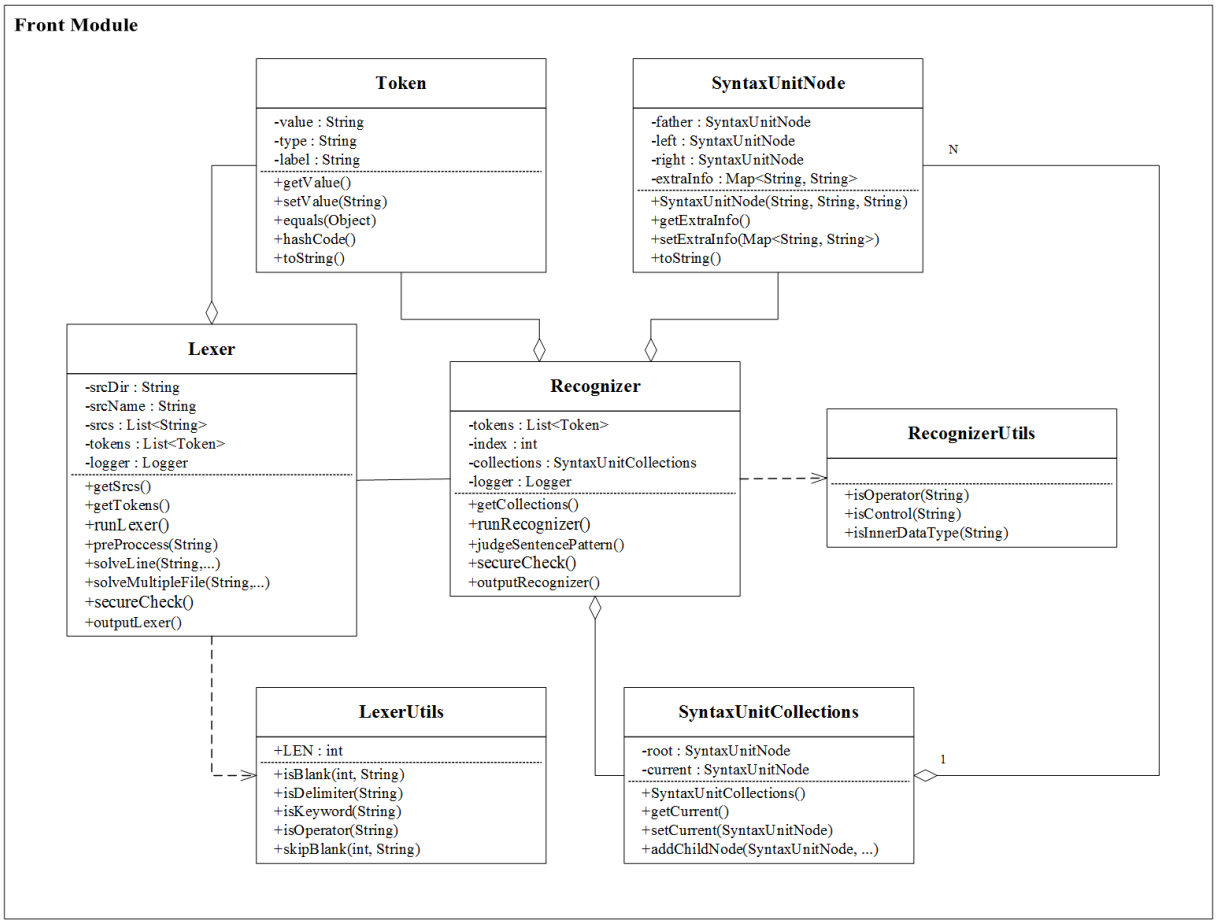

图 15　编译前端模块类图

图 15 中，Lexer 类调用 LexerUtils 类中的方法，所以二者是依赖的关系。同理，Recognizer 类和 RecognizerUtils 类之间也是依赖关系。Lexer 类输出的结果 tokens 作为了 Recognizer 类的输入，所以二者是关联关系。其它的如 Token、SyntaxUnitNode 等类，是核心类中存储数据的部分，所以与核心类之间是聚合的关系。注意，本文所有的类图中出于篇幅的考虑只列出了部分属性和公有的、核心的方法。下面将简要介绍一下各类的成员情况。

Lexer 类是词法分析的核心类，其成员说明如下表 20 所示。

表 20　Lexer 类成员说明

成员	说明
srcDir	源文件所在目录
srcName	源文件名称
srcs	源代码存储的列表（预处理后的源代码）
tokens	词法分析生成的单词流列表
logger	日志记录

getSrcs()	外界获取源代码列表的接口。
getTokens()	外界获取单词流列表的接口。
runLexer()	词法分析主体，调用其它处理函数完成词法分析整个过程。
preProccess(String)	读入源代码文件，对其进行预处理把结果存储在 srcs 中。
solveLine(String,...)	对输入行进行匹配识别其中的单词，同时生成此行对应的编号。
solveMultipleFile(String,...)	处理包含多个源代码文件的情况。
secureCheck()	对源代码是否符合安全 C 规范进行集中检测，只能检测部分规则，其它规则分散实现在前端代码中。
outputLexer()	输出整个词法分析模块。

Recognizer 类的功能是从输入的单词流中识别出文法单元，传递给编译后端模块使用，其成员说明如下表 21 所示。

表 21　Recognizer 类成员说明

成员	说明
tokens	输入的单词流列表
index	当前遍历到单词的索引
collections	输出的文法单元集合
logger	日志记录
getCollections()	外界获取文法单元集合的接口。
runRecognizer()	文法单元识别的主体，遍历输入读入单词流使用下推自动算法从中识别出文法单元，并加入到文法单元集合中。
judgeSentencePattern()	表示判断当前分析栈中的单词可能构成的文法单元的类型。
secureCheck()	同词法分析中的安全 C 检验，这里主要检验源代码中是否含有递归函数。
outputRecognizer()	输出文法单元识别模块。

Token 类定了识别出的单词的存储模板，其成员说明如下表 22 所示。

表 22　Token 类成员说明

成员	说明
value	单词的字符串值
type	单词的字符串类型（如整数、标识符等）
label	单词的编号

getValue()	获取单词的字符串值（属性的获取方法，其它属性略去）。
setValue(String)	设置单词的字符串值（属性的设置方法，其它属性略去）。
equals(Object)	自定义 Token 类相等的方法，重写了 Object 类中的默认 equals 方法。
hashCode()	自定义 Token 类生成哈希值的方法，重写了 Object 类中的默认 hashCode 方法。
toString()	自定义 Token 类的字符串表示。

SyntaxUnitNode 类定义了构成文法单元节点的模板，其成员说明如下表 23 所示。

表 23　SyntaxUnitNode 类成员说明

成员	说明
father	当前文法单元节点的父节点
left	当前文法单元节点的左节点
right	当前文法单元节点的右节点
extraInfo	记录的其它属性信息（如节点的类型等）
SyntaxUnitNode(…)	文法单元节点的构造函数。
getExtraInfo()	获取其它属性信息（属性的获取方法，其它属性略去）。
setExtraInfo(…)	设置其它属性信息（属性的设置方法，其它属性略去）。
toString()	自定义 SyntaxUnitNode 类的字符串表示。

SyntaxUnitCollections 类定义了存储文法单元的集合的模板，其成员说明如下表 24 所示。

表 24　SyntaxUnitCollections 类成员说明

成员	说明
root	指向文法单元集合最上面的元素
current	指向文法单元集合当前的元素
SyntaxUnitCollections()	新定义一个空的文法单元集合。
getCurrent()	获取当前的元素（属性的获取方法，其它属性略去）。
setCurrent(…)	设置当前的元素（属性的设置方法，其它属性略去）。
addChildNode(…)	向集合中增添一个文法单元。

最后，LexerUtils 类和 RecognizerUtils 类主要包含了一些具有公共功能的静态方法，如跳过字符流中的空格，判断单词是否是关键字，判断单词是否是内置类型等，比较简单。

5.2.2 编译后端模块

编译后端模块会遍历输入的文法单元集合并基于文法单元的形式验证结果和目标机器的指令集输出目标代码。该模块的主体是 Assembler 类，其任务是把输入的文法单元集合转变为目标机器代码。AssemblerExpression 类是专门处理表达式文法单元的类，其简化了 Assembler 类的实现。LexerUtils 辅助核心类工作的工具类，包含一些静态变量和静态方法。AssemblerDTO 是一个数据传输类，封装了目标机器的语境信息。AssemblerFileHandler 类维护着生成的目标代码，可以生成目标代码文件。下面将给出编译后端模块的类图，如图 16 所示。

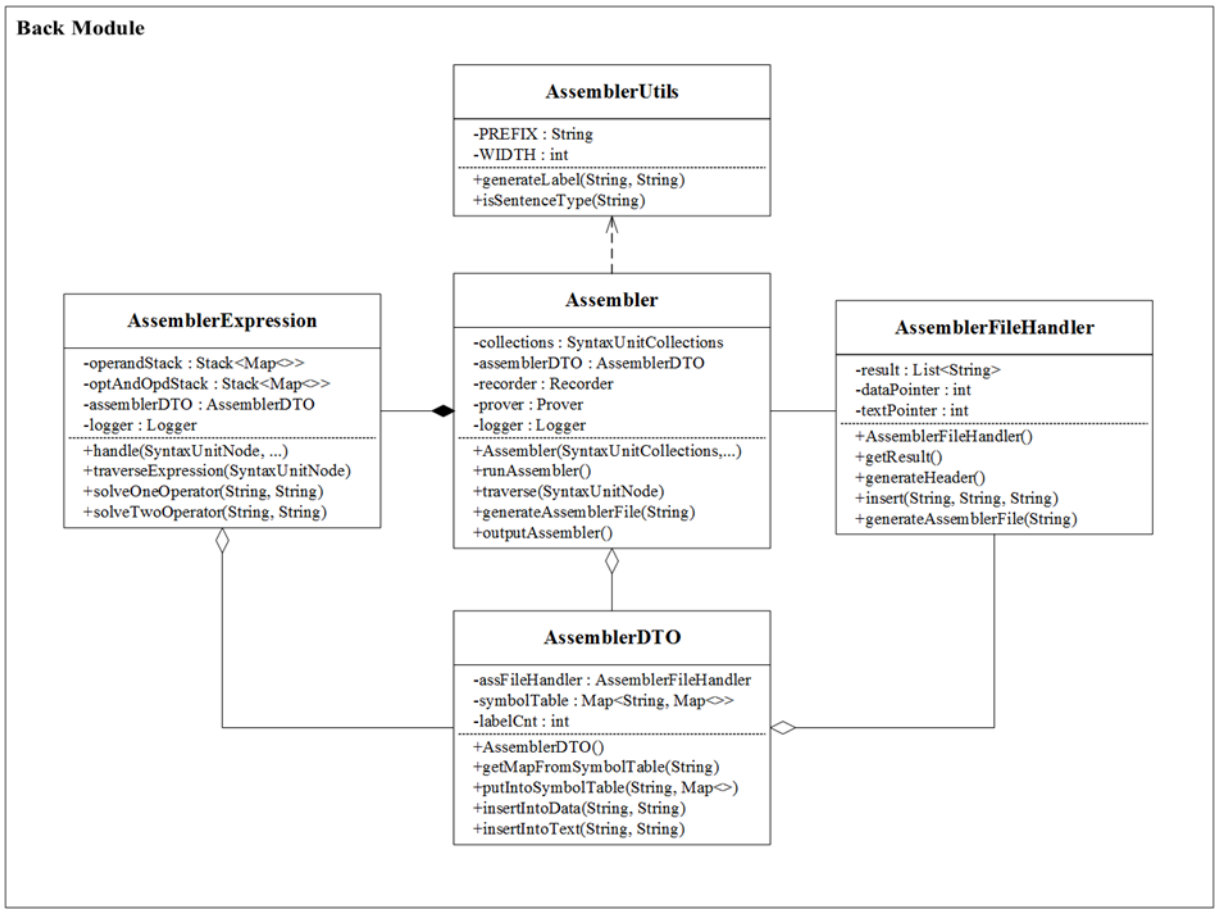

图 16　编译后端模块类图

图 16 中，Assembler 类调用了 LexerUtils 类中的方法，故二者是依赖关系。AssemblerExpression 类是从 Assemble 类中独立出来的部分函数所构成的，故二者是聚合关系。AssemblerDTO 类是 Assembler 类和 AssemblerExpression 类共享数据的类，故与二者都是组合的关系。AssemblerFileHandler 类是 AssemblerDTO 类中的变量，故二者是组合的关系。下面将简要介绍一下各类的成员情况。

Assembler 类是后端目标代码生成的核心类，其成员说明如下表 25 所示。

表 25　Assembler 类成员说明

成员	说明
collections	输入的文法单元集合
assemblerDTO	数据传输类实例
recorder	记录类实例
prover	验证器实例
logger	日志记录
Assembler(...)	新建一个目标码生成器。
runAssembler()	目标码生成模块的主体，主要功能是把输入的文法单元集合转化为与机器指令集相关的目标代码。
traverse(SyntaxUnitNode)	遍历文法单元的节点。
generateAssemblerFile(…)	生成目标代码文件。
outputAssembler()	输出整个目标代码模块。

AssemblerExpression 类是 Assembler 类功能的补充，其成员说明如下表 26 所示。

表 26　AssemblerExpression 类成员说明

成员	说明
operandStack	操作数栈（分析栈）
optAndOpdStack	操作数操作符栈（二者同时放到一个栈中）
assemblerDTO	数据传输类实例
logger	日志记录
handle(SyntaxUnitNode,....)	接收 Assembler 类传递的表达式文法单元，并进行处理。
traverseExpression(…)	遍历表达式文法单元的节点。
solveOneOperator(…)	处理含有单目运算符的表达式文法单元。
solveTwoOperator(…)	处理含有双目运算符的表达式文法单元。

AssemblerDTO 中定义了存储目标机语境信息的模板，其成员说明如下表 27 所示。

表 27　AssemblerDTO 类成员说明

成员	说明
assFileHandler	目标代码存储类实例
symbolTable	符号表
labelCnt	标签个数

AssemblerDTO()	新建一个数据传输类实例。
getMapFromSymbolTable(…)	从符号表中获取数据。
putIntoSymbolTable(…)	向符号表中存储数据。
insertIntoData(String, String)	向目标代码列表的数据域插入指令。
insertIntoText(String, String)	向目标代码列表的文本域插入指令。

AssemblerFileHandler 类中存储后端模块生成的目标代码，其成员说明如下表 28 所示。

表 28　AssemblerFileHandler 类成员说明

成员	说明
result	目标代码列表
dataPointer	目标代码数据域指针
textPointer	目标代码文本域指针
AssemblerFileHandler()	新建一个目标码处理类实例。
getResult()	外界获取目标码列表的接口。
generateHeader()	产生目标代码的头部指令。
insert(String, String, String)	向目标代码列表中插入一条指令。
generateAssemblerFile(String)	生成目标代码文件。

最后，AssemblerUtils 类包含了一些简单的具有工具性质的静态属性和方法，如每行目标代码的前缀和长度、生成标号的方法等。

5.2.3 形式验证模块

形式验证模块的主要任务是对输入的文法单元和目标码模式进行语义的形式验证，并向编译后端模块返回验证结果。该模块的主体是 Prover 类，根据目标码模式内是否含有循环结构调用相应的算法完成证明过程。PropositionMappingAlgorithm 类封装的是命题映射算法的实现，DerivationAlgorithm 类封装的是自动推理算法的实现，LoopInteractiveProving-Algorithm 类则是循环交互证明算法的实现。DerivationDTO 类存储着证明过程的中间结果。ProverUtils 类是一个辅助的工具类。下面将给出形式验证模块的类图，如图 17 所示。

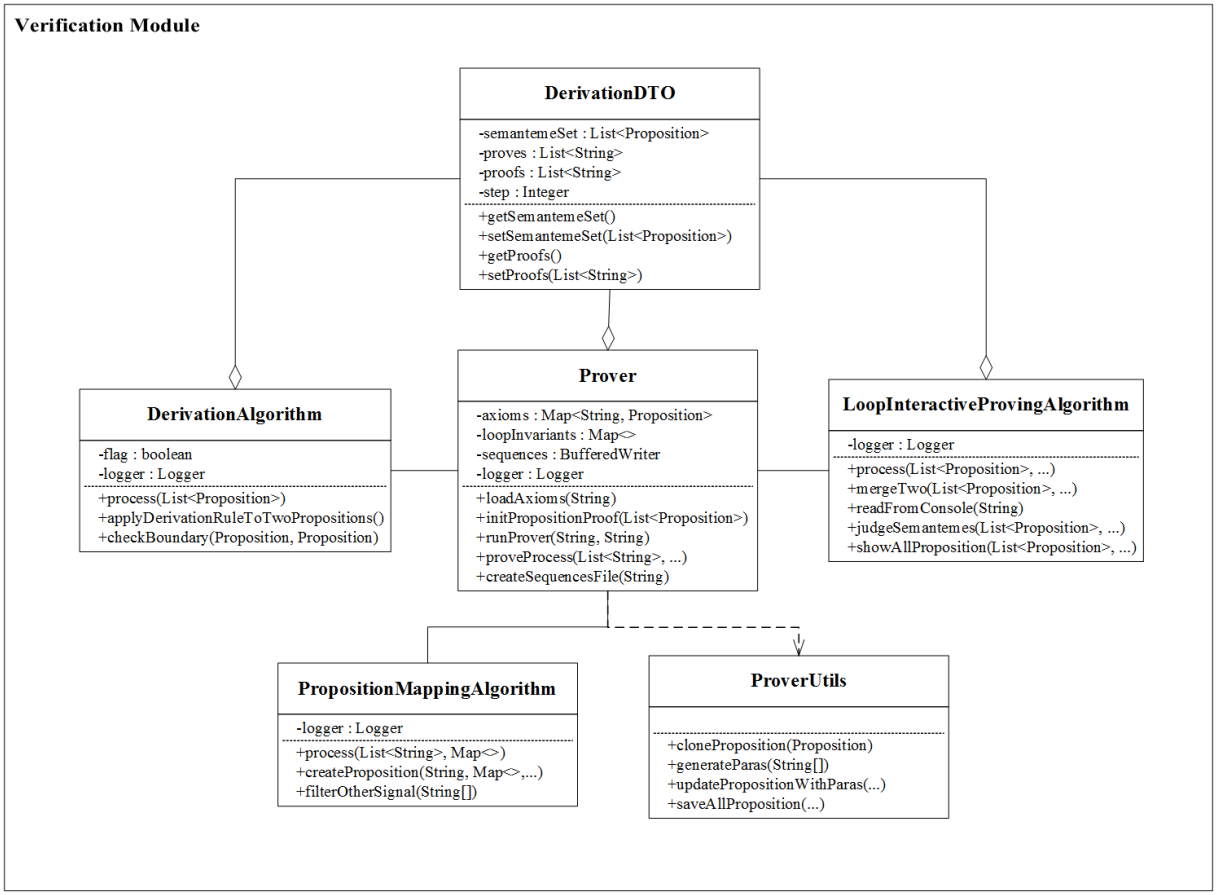

图 17　形式验证模块类图

图 17 中，Prover 类与三个算法实现类是关联的关系，且其调用了 ProverUtils 中的方法，故 Prover 类与 ProverUtils 类之间是依赖的关系。DerivationDTO 类存储着证明过程中的结果，是 Prover 类和左右两个算法实现类传递信息的媒介，故 DerivationDTO 类与两个算法实现类之间是组合的关系。下面将简要介绍一下各类的成员情况。

Prover 类是形式验证模块的核心类，其成员说明如下表 29 所示。

表 29　Prover 类成员说明

成员	说明
axioms	专用公理集
loopInvariants	循环不变式
sequences	证明序列
logger	日志记录
loadAxioms(String)	从文件中导入专用公理。
initPropositionProof(…)	生成证明的前提集。
runProver(String, String)	初始化相关参数，启动证明器。

proveProcess(...)	证明的主体过程。
createSequencesFile(String)	生成证明序列文件。

PropositionMappingAlgorithm 定义了命题映射算法的实现模板，其成员说明如下表 30 所示。

表 30　PropositionMappingAlgorithm 类成员说明

成员	说明
logger	日志记录
process(List<String>,…)	算法处理的主体，把输入的目标码模式转化为命题的形式。
createProposition(String,...)	把当前遍历到的目标码转化为一个命题。
filterOtherSignal(String[])	去除当前遍历到的目标码中其它的符合。

DerivationAlgorithm 定义了自动推理算法的实现模板，其成员说明如下表 31 所示。

表 31　DerivationAlgorithm 类成员说明

成员	说明
flag	表示推理中的两个命题是否有关联
logger	日志记录
process(List<Proposition>)	对目标码模式命题集合进行推理，输出存储推理结果的 DerivationDTO 类实例。
applyDerivationRuleToTwoPropositions()	选择推理规则对当前的两个命题进行推理。
checkBoundary(Proposition, Proposition)	检查输入的两个命题是否合法。

LoopInteractiveProvingAlgorithm 定义了循环交互算法的实现模板，其成员说明如下表 32 所示。

表 32　LoopInteractiveProvingAlgorithm 类成员说明

成员	说明
logger	日志记录
process(...)	对目标码模式命题集合进行归纳推理，输出最终结果。
mergeTwo(...)	把两个中间状态的目标码模式命题集合进行合并。
readFromConsole(String)	从控制台读入用户输入的循环不变式。
judgeSemantemes(...)	判断文法单元和目标码模式的语义是否一致。
showAllProposition(...)	输出目标码模式命题集合。

最后，DerivationDTO 类定义了存储中间结果的模板，主要包括证明序列、证据和

证明步数等属性和相应的 get/set 方法。ProverUtils 类主要包括一些静态的工具方法，如从当前命题克隆出一个新的命题、使用参数更新命题等。

5.2.4 用户界面模块

用户界面模块是用户与程序交互的接口，其主要任务是接受用户的输入，反馈运行的实时信息，并把程序运行的最终结果显示出来。该模块的主体是 MainWindow 类，在其构造函数中会生成整个界面的所有部分。CustomPanel 类继承自 JPanel 类，表示自定义的面板。CustomButtonUI 类继承自 BasicButtonUI 类，表示自定义的按钮界面。CustomTreeCellRenderer 类继承自 DefaultTreeCellRenderer 类，表示 JTree 类的自定义单元绘制器。Node 表示 JTree 类的节点值类型。JPanel 类、BasicButtonUI 类、DefaultTreeCellRenderer 类和 JTree 类都是 Java 用户界面开发工具包 Swing 的内置组件。下面将给出用户界面模块类图，如图 18 所示。

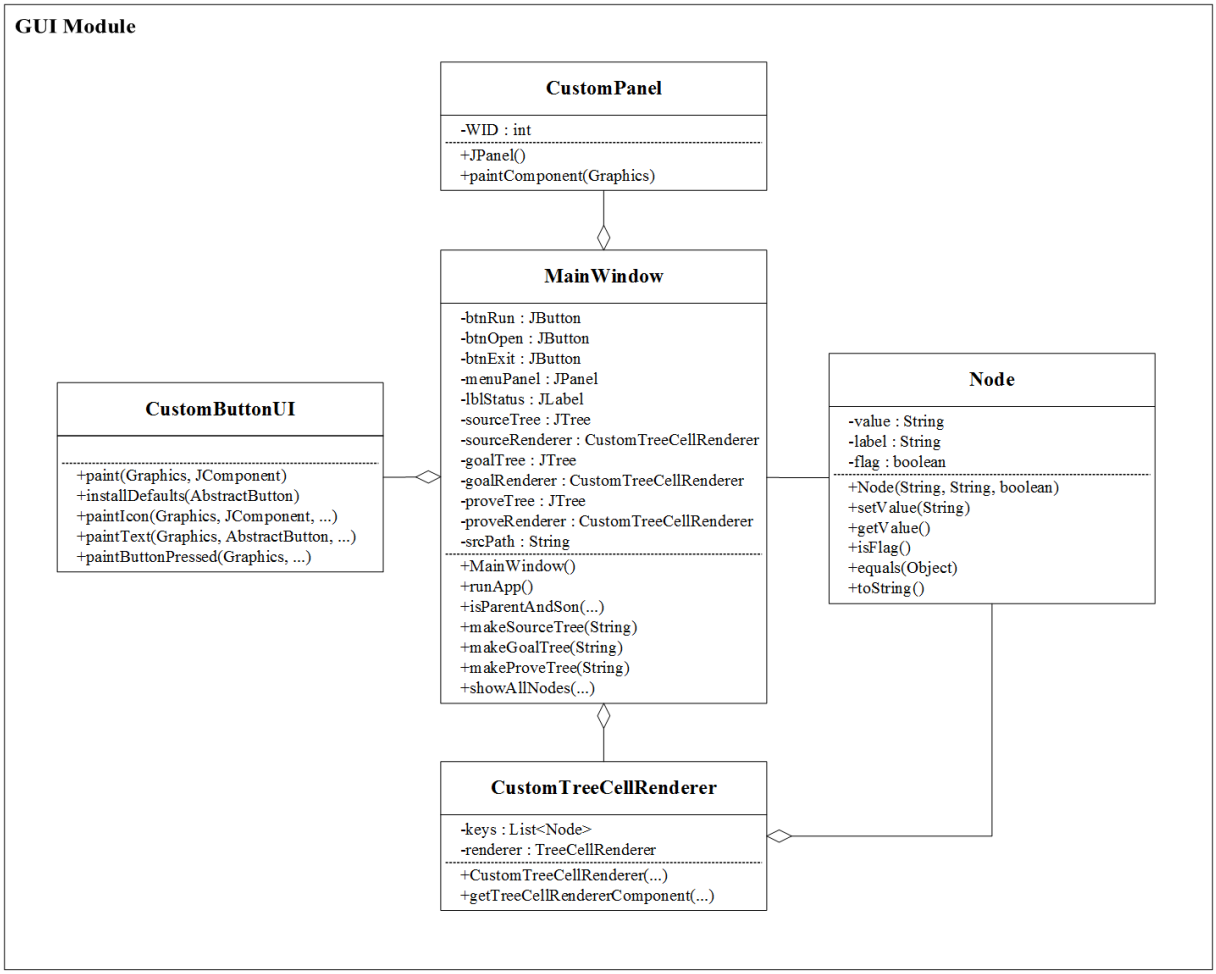

图 18　用户界面模块类图

图 18 中，MainWindow 类包含有 CustomPanel 类、CustomButtonUI 类和 CustomTree-CellRenderer 类的实例，故 MainWindow 类与它们是组合的关系。

CustomTreeCellRenderer 类中含有一个由 Node 类的实例组成的列表，故二者也是组合的关系。Node 类实例是在 MainWindow 类中构造的，故二者是关联的关系。下面将简要介绍一下各类的成员情况。

MainWindow 类是用户界面模块的主体窗口类，其成员说明如下表 33 所示。

表 33　MainWindow 类成员说明

成员	说明
btnRun	运行按钮
btnOpen	打开按钮
btnExit	退出按钮
menuPanel	菜单栏
lblStatus	状态栏
sourceTree	源代码树
sourceRenderer	源代码树节点绘制器
goalTree	目标代码树
goalRenderer	目标代码树节点绘制器
proveTree	证明序列树
proveRenderer	证明序列树节点绘制器
srcPath	源代码路径
MainWindow()	新建一个程序界面，初始化界面中的所有组件。
runApp()	启动程序，调用编译前端、编译后端和形式验证模块的相关核心类。
isParentAndSon(...)	判断树上的两个节点是否是父子关系。
makeSourceTree(String)	构造源代码树。
makeGoalTree(String)	构造目标代码树。
makeProveTree(String)	构造证明序列树。
showAllNodes(...)	显示指定树种所有节点的值。

Node 类定义了 JTree 组件上每个节点值的模板，其成员说明如下表 34 所示。

表 34　Node 类成员说明

成员	说明
value	一个字符串类型的值，代表着源代码、目标代码或证明序列集合中的一行

label	编号
flag	是否显示编号
Node(String, String, …)	新建一个节点。
setValue(String)	设置节点的字符串值（属性的设置方法，其它属性略去）。
getValue()	获取节点的字符串值（属性的获取方法，其它属性略去）。
isFlag()	判断是否显示 value 的编号。
equals(Object)	定义两个节点相等的方法，这里是通过编号来判断的。
toString()	自定义 Node 类的字符串表示。

最后，CustomPanel 类、CustomButtonUI 类和 CustomTreeCellRenderer 类都是继承自 Swing 工具包中的内置类，只是结合实际需要修改了一些方法的默认实现，但方法的作用保持不变，故在此不再赘述。

5.3 系统实验

5.3.1 实验环境

硬件环境：

（1）主机 CPU：Intel Core i5 CPU 2.7GHz

（2）内存：4GB（推荐 1GB 以上）

（3）磁盘空间：空闲空间 8GB（推荐 4GB 以上）

软件环境：

（1）操作系统：Mac OS X 10.12 或 Windows 7（或以上）

（2）软件：JDK1.8 版（或以上）

5.3.2 实验过程与分析

本节通过一个例子来展示编译验证系统工具的工作过程，其中输入的源代码如附录 II 中的 test2.c 所示。源代码的主要功能是计算一定范围内奇数和的两倍及一个整数的平方根，包含的基本的语句有 if 语句、while 语句、do-while 语句和 for 语句等，虽然比较简单但包含的文法单元较为全面，可以较好的反映本文所提出的的方法及编译验证系统的工作流程。经过本课题实现的工具对源代码的编译验证，所得到的主要输出文件如下图 19 所示。

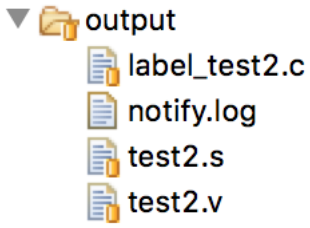

图 19　编译验证工具输出

图 19 中，label_test2.c 表示源文件进行层级编码后的输出，如附录 III 所示。test2.s 为系统输出的目标码文件，test2.v 表示形式验证模块输出的证明序列文件。notify.log 文件记录了编译验证过程中的一些日志信息，主要用作系统的调试和追踪。下面给出目标码文件的部分片段，如表 35 所示。

表 35　目标码片段

lwz 0,8(31)	# 3.7.1_ex
li 9,2	# 3.7.1_ex
divw 11,0,9	# 3.7.1_ex
mullw 9,11,9	# 3.7.1_ex
subf 0,9,0	# 3.7.1_ex
stw 0,24(31)	# 3.7.1_ex
lwz 0,24(31)	# 3.7.1_ex
li 9,0	# 3.7.1_ex
cmp 7,0,0,9	# 3.7.1_ex
li 0,0	# 3.7.1_ex
li 9,1	# 3.7.1_ex
isel 0,9,0,30	# 3.7.1_ex
stw 0,28(31)	# 3.7.1_ex
lwz 0,28(31)	# 3.7.1_if
cmpi 7,0,0,0	# 3.7.1_if
beq 7,.L2	# 3.7.1_if
lwz 9,8(31)	# 3.7.1.1_ex
li 0,2	# 3.7.1.1_ex
mullw 0,9,0	# 3.7.1.1_ex
stw 0,24(31)	# 3.7.1.1_ex
lwz 9,16(31)	# 3.7.1.1_ex
lwz 0,24(31)	# 3.7.1.1_ex

```
        subf 0,9,0                                  # 3.7.1.1_ex
        stw 0,28(31)                                # 3.7.1.1_ex

        lwz 0,28(31)                                # 3.7.1.1_as
        stw 0,16(31)                                # 3.7.1.1_as

.L2:                                                # 3.7.1_if
```

表 35 中，每行代码末尾编号都是以相同前缀# 3.7.1 开始的，而# 3.7.1 编号正是源代码中<if-statement>所在行开始的编号，这说明上述代码段代表 if 语句及其内的子句。编号的末尾用两个字符区分语句的类型，因为源代码中某一行可能对应着多个文法单元的部分结构。下面将给出 if 文法单元的证明序列的片段，如表 36 所示。

表 36　证明序列片段

```
if : 3.7.1_if
==============目标码模式命题===============
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>
P5 = .L1:

==============推导序列===============
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>        P4
S7 = .L1:        P5
S8 = (<LOG-EXP> < 0 -> PC = PC + 4 || <LOG-EXP> > 0 -> PC = PC + 4 || <LOG-EXP> == 0 -> PC =
PC + @.L1) ∧ (<STA-LIST>) ∧ (.L1:)        S5, S6, S7, CI
S9 = (<LOG-EXP> != 0 -> <STA-LIST> || <LOG-EXP> == 0 -> null)            S8, REDUCE
S10 = (<LOG-EXP> != 0 -> σ(<STA-LIST>) || <LOG-EXP> == 0 -> skip)            S9, σ
```

```
=============前置语义================
<LOG-EXP> != 0 -> σ(<STA-LIST>) || <LOG-EXP> == 0 -> skip

===============结论================
前置语义和推理出的语义是否一致 ：
true
if 语句验证结果 ： 验证成功
```

表 36 中，目标码模式的证明过程主要分为三步，首先给出目标码模式的命题作为前提集，如目标码模式命题 P1~P5 所示。然后依据本文公理系统的推理规则和公理对目标码模式命题集进行推理，并进行语义获取操作得到目标码模式的语义，如推导序列所示。最后把目标码模式的语义与文法单元的前置语义进行对比，若二者语义一致则输出 true；反之，输出 false。下面将给出编译验证系统的运行时界面，如图 20 所示。

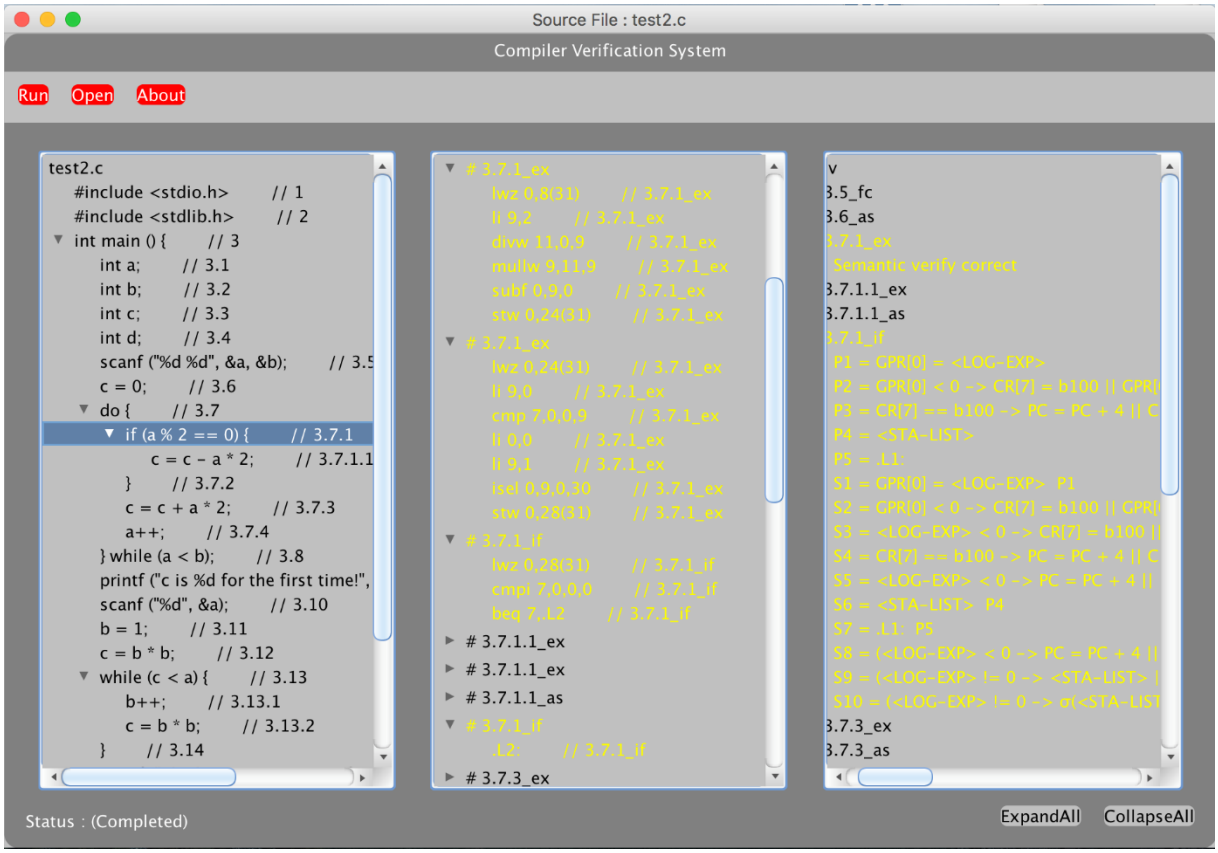

图 20　系统运行界面图

图 20 中，系统界面由菜单栏、主体和状态栏三个部分组成。菜单栏包括一个启动系统运行的“Run”按钮、一个打开指定输入文件的“Open”按钮和提供帮助的“About”按钮，状态栏包括左边的状态显示和右边控制主体面板展开和折叠的“ExpandAll”和“CollapseAll”按钮。主体部分为三个滚动面板，分别用来显示编码后的源代码、目标

代码和证明序列。可以看到，当选定 if 文法单元所在的开始行时，目标代码面板和证明序列面板会有相应的代码从折叠状态展开并高亮显示。若选定目标代码面板或者证明序列面板中的某些行，也会在剩余面板中展开并显示对应的行。这些都表明本文的编译验证系统成功地实现了源代码到目标代码及证明序列的相互之间的可追踪性需求，可以作为一个追踪工具提供给用户使用。

5.4 小结

本章介绍了编译验证工具的设计与实现，具体如下：

（1）设计了编译验证工具的系统架构并给出整体框架图。系统框架由编译前端模块、编译后端模块、形式验证模块和用户界面模块四个部分组成。其中，编译前端模块和编译后端模块主要完成对源代码的编译处理，并把从中识别出的文法单元交给形式验证模块进行形式验证，最终生成的结果由用户界面模块显示出来。

（2）根据编译验证系统的整体框架图，本文给出了其四个组成模块的类图，并通过把类图转换为实际的类从而构建出了整个系统。在工具的实现部分，主要讨论了每个模块类的组成和相互之间的关系，并对核心的类作出了较为详细的介绍，可以看到核心类的方法与本文提出的编译验证技术的相关部分是一一对应的。

（3）系统实验部分选择一个典型源代码文件作为工具的输入，经过系统的编译处理和形式证明最终得到了带有层级编号的目标代码和证明序列文件。上述过程保证了源文件编译过程的正确性、安全性和可追踪性，从而最终完成了本文的研究目标。

结论

论文总结

随着计算机应用技术的快速发展，计算机软件已经深深融入到了航空航天领域之中，现代飞机几乎所有重要功能系统都与机载软件密切相关。编译器作为机载软件开发过程中的重要工具，负责将源程序翻译成目标程序，是实现软件从设计到在硬件上运行的桥梁。传统编译存在难以满足安全关键领域开发标准、难以分析和检验其正确性和安全性、验证过程复杂繁琐以及检测覆盖不全面等问题，本课题通过对编译系统相关构建和验证方法的深入研究，提出了基于文法单元和目标码模式的编译语义验证方法，并对现有的编译技术进行了一定的扩展使其能支持本文所提出的形式验证方法，还实现了一套遵循 DO-178C 标准的编译验证系统工具集。本文的主要研究成果如下：

（1）提出了基于文法单元和目标码模式的编译语义验证方法，把编译过程正确性验证等价为对每个 C 文法单元的验证。采用形式推理的技术，在本文构建的公理系统中验证编译前后每个 C 文法单元和对应的目标代码模式的语义是否一致，并给出了实现验证方法的整体架构。

（2）提出了应用于编译形式化验证过程中的三大算法。命题映射算法的作用是把目标码模式转换为命题的形式，便于后续的推理和证明；自动推理算法是验证过程的核心，它基于本文构建的公理系统，使用前提集和推理规则推导出一系列新命题，把这些新的命题作为定理加入前提中进行后续的证明；循环交互证明算法主要用于含有循环结构的文法单元的证明，其理论基础是限定数学归纳法。

（3）提出了安全 C 编译器构建中的关键方法，并给出了系统架构。为了支持编译语义验证方法，采用了有限自动机和下推自动机作为编译前端的主体实现算法，解决了词法记号和文法单元的识别问题。使用层级编码和安全 C 检验方法实现了 DO178C 规定的 A 级编译验证系统的可追踪性和高安全性需求。

最后，本课题中的编译验证系统是一个很好的工具原型成功地实现了本文所提出的编译构建和验证方法并证明了这些方法的有效性，解决了符合 DO178C 标准的 A 级编译系统的正确性验证和构造问题。在此系统的基础上进行功能的不断丰富和完善，最终可以得到一个成熟的安全关键软件编译系统。

工作展望

目前的编译验证系统工具虽然成功的实现了编译和验证功能，符合了安全关键领域的相关规范，但依旧存在不足的地方，需要进一步改进和完善。未来的工作可以考虑以下几个方面：

（1）系统的编译功能还需要完善。虽然已经支持了基本数据类型和语句的编译，但对于结构体的支持还没有实现，数组的处理也不是很好，下一步需要重点在这两个方向上努力。

（2）安全 C 检验是系统本身一个非常重要的功能，本文在实现时把检验过程穿插到了编译前端的多个处理阶段中，这不利于系统代码的管理和扩展。后面若有新的安全 C 规则需要加入或修改时，对系统代码的改动会比较大。未来需要对安全 C 规则作出一定的形式化和规范化，并把安全 C 检验过程重构为一个独立的可扩展的模块。

（3）本文实现的系统编译部分由前端和后端两个模块组成，缺少了优化了部分。优化部分通过改进中间代码形式可以使后端产生效率更高的代码，优化对程序的语义没影响。未来可能需要在系统的编译部分加入优化模块。

附录

附录 I　安全 C 子集文法

```
<source files>::= <header-file-list> <macro-definition-list> <type-definition-list>
                  <function-prototype-list> <extern-variable-list> <global-variable-list>
                  <main-function> <function-list>
<header-file-list>::= <header-file> <header-file-list> | <header-file>
<header-file>::= #include '< <file-name> '> | #include " <file-name> "
<file-name>::= <IDENTIFIER>.h
<macro-definition-list>::= <macro-definition> <macro-definition-list> | <macro-definition>
<macro-definition>::= #define <macro-name> <macro-text>
<macro-name>::= <IDENTIFIER>
<macro-text>::= <IDENTIFIER>
<type-definition-list>::= <type-definition> <type-definition-list> | <type-definition>
<type-definition>::= struct <IDENTIFIER> '{ <struct-declaration-list> '} ;
<struct-declaration-list>::= <struct-declaration> <struct-declaration-list> | <struct-declaration>
<struct-declaration>::= <specifier-qualifier-list> <struct-declarator-list> ;
<specifier-qualifier-list>::= <type-specifier> <specifier-qualifier-list> | <type-specifier> |
                              <type-qualifier> <specifier-qualifier-list> | <type-qualifier>
<type-specifier>::= void | short | int | long | float | double | signed | unsigned |
                    <struct-specifier> | <union-specifier> | <enum-specifier>
<struct-specifier>::= struct <IDENTIFIER> '{ <struct-declaration-list> '} |
                      struct '{ <struct-declaration-list> '} |
                      struct <IDENTIFIER>
<union-specifier>::= union <IDENTIFIER> '{ <struct-declaration-list> '} |
                      union '{ <struct-declaration-list> '} |
                      union <IDENTIFIER>
<enum-specifier>::= enum <IDENTIFIER> '{ <enumerator-list> '} |
                    enum '{ <enumerator-list> '} |
                    enum <IDENTIFIER>
<struct-declarator-list>::= <struct-declarator> , <struct-declarator-list> | <struct-declarator>
<struct-declarator>::= <declarator> | <declarator> : <primary-expression> | : <primary-expression>
<declarator>::= <pointer> <direct-declarator> | <direct-declarator>
<enumerator-list>::= <enumerator> , <enumerator-list> | <enumerator>
<enumerator>::= <IDENTIFIER> | <IDENTIFIER> = <primary-expression>
<pointer>::= * | * <type-qualifier-list> | *<pointer> | * <type-qualifier-list> <pointer>
<type-qualifier-list>::= <type-qualifier> <type-qualifier-list> | <type-qualifier>
<direct-declarator>::= <IDENTIFIER> |
                       '( <declarator> ') |
                       <direct-declarator> '[ <primary-expression> '] |
                       <direct-declarator> '[ '] |
                       <IDENTIFIER> '( <parameter-type-list> ') |
                       <IDENTIFIER> '( <identifier-list> ') |
                       <IDENTIFIER> '( ')
```

```
<direct-declarator>::= <IDENTIFIER> <direct-declarator-right> |
                       '( <declarator> ') <direct-declarator-right>|
                       <IDENTIFIER> '( <parameter-type-list> ') <direct-declarator-right> |
                       <IDENTIFIER> '( <identifier-list> ') <direct-declarator-right> |
                       <IDENTIFIER> '( ') <direct-declarator-right> |
                       <direct-declarator-right>
<direct-declarator-right>::= '[ <primary-expression> '] <direct-declarator-right> |
                             '[ '] <direct-declarator-right> |
                             <empty>
<empty>::= NULL
<parameter-type-list>::= <parameter-list> | <parameter-list> , ...
<parameter-list>::= <parameter-declaration> , <parameter-list> | <parameter-declaration>
<parameter-declaration>::= <declaration-specifier-list> <declarator> |
                           <declaration-specifier-list> <abstract-declarator> |
                           <declaration-specifier-list>
<identifier-list>::= <IDENTIFIER> , <identifier-list> | <IDENTIFIER>
<declaration-specifier-list>::= <storage-class-specifier> <declaration-specifier-list> |
                                <storage-class-specifier> |
                                <type-specifier> <declaration-specifier-list> |
                                <type-specifier> |
                                <type-qualifier> <declaration-specifier-list> |
                                <type-qualifier>
<storage-class-specifier>::= auto | register | static | extern | typedef
<type-qualifier>::= const | volatile
<abstract-declarator>::= <pointer> | <direct-abstract-declarator> | <pointer> <direct-abstract-declarator>
<direct-abstract-declarator>::= '( <abstract-declarator> ') |
                                '( <parameter-list> ') |
                                <direct-abstract-declarator> '( <parameter-list> ') |
                                <direct-abstract-declarator> '( ') |
                                '( ') |
                                '[ <primary-expression> '] |
                                <direct-abstract-declarator> '[ <primary-expression> '] |
                                <direct-abstract-declarator> '[ '] |
                                '[ ']
<direct-abstract-declarator>::= '( <abstract-declarator> ') <direct-abs-declarator-right> |
                                '( <parameter-list> ') <direct-abs-declarator-right> |
                                '( ') <direct-abs-declarator-right> |
                                '[ <primary-expression> '] <direct-abs-declarator-right> |
                                '[ '] <direct-abs-declarator-right> |
                                <direct-abs-declarator-right>
<direct-abs-declarator-right>::= '( <parameter-list> ') <direct-abs-declarator-right> |
                                 '( ') <direct-abs-declarator-right> |
                                 '[ <primary-expression> '] <direct-abs-declarator-right> |
                                 '[ '] <direct-abs-declarator-right> |
```

<empty>
<function-prototype-list>::= <function-prototype> <function-prototype-list> | <function-prototype>
<function-prototype >::= <type-specifier> <funtion-name> '(<parameter-type-list> ') ;
<funtion-name>::= <IDENTIFIER>
<extern-variable-list>::= <extern-variable> <extern-variable-list> | <extern-variable>
<extern-variable >::= extern <type-specifier> <variable-name-list> ;
<variable-name-list>::= <variable-name> , <variable-name-list> | <variable-name>
<variable-name>::= <IDENTIFIER>
<global-variable-list>::= <global-variable> <global-variable-list> | <global-variable>
<global-variable>::= <type-specifier> <variable-name-list> ; |
<type-specifier> <assignment-expression> ;
<main-function>::= void main '(') '{ <declaration-list> <statement-list> '}
<declaration-list>::= <declaration> <declaration-list> | <declaration>
<declaration>::= <declaration-specifier-list> <init-declarator-list> ; | <declaration-specifier-list> ;
<init-declarator-list>::= <init-declarator> , <init-declarator-list> | <init-declarator>
<init-declarator>::= <declarator> | <declarator> = <initializer>
<initializer>::= <assignment-expression> | '{ <initializer-list> '} | '{ <initializer-list> , '}
<initializer-list>::= <initializer> , <initializer-list> | <initializer>
<function-list>::= <function> <function-list> | <function>
<function>::= <type-specifier> <function-name> '(<parameter-type-list> ') '{ <declaration-list>
<statement-list> '}
<function-name>::= <IDENTIFIER>

<statement-list>::= <statement> <statement-list> | <statement>
<statement>::= <if-statement> | <switch-statement> | <while-statement> |
<do-statement> | <for-statement> | <jump-statement> |
<expression-statement>
<if-statement>::= if '(<logical-expression> ') <statement> |
if '(<logical-expression> ') <statement> else <statement> |
if '(<logical-expression> ') '{ <statement-list> '} |
if '(<logical-expression> ') '{ <statement-list> '} else '{ <statement-list> '}
<switch-statement>::= switch '(<expression> ') '{ <case-statement-list> <default-statement> '}
<case-statement-list>::= <case-statement> <case-statement-list> | <case-statement>
<case-statement>::= <case-constant-expression> : <statement-list>
<case-constant-expression> ::= case <IDENTIFIER>
<default-statement>::= default : <statement-list>
<while-statement>::= while '(<logical-expression> ') '{ <statement-list> '}
<do-statement>::= do '{ <statement-list> '} while '(<logical-expression> ')
<for-statement>::= for '(<expression> ; <expression> ; <expression> ') '{ <statement-list> '}
<jump-statement>::= <break-statement> | <continue-statement> | <return-statement>
<break-statement>::= break ;
<continue-statement>::= continue ;
<return-statement>::= return <expression> ; | return ;
<expression-statement>::= <empty-statement> | <expression> ;

```
<empty-statement>::= ;

<expression>::= <arithmetic-expression> | <relational-expression> | <logical-expression> |
                <assignment-expression> | <conditional-expression> | <argument-expression> |
                <primary-expression>
<arithmetic-expression>::= <primary-expression> + <expression> |
                           <primary-expression> - <expression> |
                           <primary-expression> * <expression> |
                           <primary-expression> / <expression> |
                           <primary-expression> % <expression>
<relational-expression>::= <primary-expression> '< <expression> |
                               <primary-expression> '> <expression> |
                               <primary-expression> '<= <expression> |
                                   <primary-expression> '>= <expression> |
                                       <primary-expression> == <expression> |
                                           <primary-expression> '!= <expression>
<logical-expression>::= <bitwise-logical-expression> | <bool-logical-expression>
<bitwise-logical-expression>::= <primary-expression> & <expression> |
                                <primary-expression> '| <expression> |
                                <primary-expression> ^ <expression> |
                                <primary-expression> '<'< <expression> |
                                <primary-expression> '>'> <expression> |
                                ~ <expression>
<bool-logical-expression>::= <primary-expression> && <expression> |
                             <primary-expression> '|'| <expression> |
                             ! <expression>
<assignment-expression>::= <variable> = <expression> |
                           <variable> += <expression> |
                           <variable> -= <expression> |
                           <variable> *= <expression> |
                           <variable> /= <expression> |
                           <variable> %= <expression> |
                           <variable> '<'<= <expression> |
                           <variable> '>'>= <expression> |
                           <variable> &= <expression> |
                           <variable> '|= <expression> |
                           <variable> != <expression> |
                           <variable> ^= <expression>
<variable>::= <IDENTIFIER>
<conditional-expression>::= <logical-expression> ? <expression> : <conditional-expression>
<argument-expression>::= <assignment-expression> , <expression>
<primary-expression>::= <IDENTIFIER> | <CONSTANT> | '( <expression> ')
```

附录 II　test2.c

```
#include <stdio.h>
#include <stdlib.h>

int main () {
	int a;
	int b;
	int c;
	int d;

	scanf ("%d %d", &a, &b);
	c = 0;
	do {
		if (a % 2 == 0) {
			c = c - a * 2;
		}
		c = c + a * 2;
		a++;
	} while (a < b);
	printf ("c is %d for the first time!", c);

	scanf ("%d", &a);
	b = 1;
	c = b * b;
	while (c < a) {
		b++;
		c = b * b;
	}
	printf ("The biggest sqrt root of %d is %d", a, b);

	// for
	for (a = 0; a < 10; a++) {
		b++;

	}

	return 0;
}
```

附录 III　label_test2.c

```
#include <stdio.h>                                  // 1
#include <stdlib.h>                                 // 2

int main () {                                       // 3
	int a;                                          // 3.1
```

```
    int b;                                          // 3.2
    int c;                                          // 3.3
    int d;                                          // 3.4

    scanf ("%d %d", &a, &b);                        // 3.5
    c = 0;                                          // 3.6
    do {                                            // 3.7
        if (a % 2 == 0) {                           // 3.7.1
            c = c - a * 2;                          // 3.7.1.1
        }                                           // 3.7.2
        c = c + a * 2;                              // 3.7.3
        a++;                                        // 3.7.4
    } while (a < b);                                // 3.8
    printf ("c is %d for the first time!", c);      // 3.9

    scanf ("%d", &a);                               // 3.10
    b = 1;                                          // 3.11
    c = b * b;                                      // 3.12
    while (c < a) {                                 // 3.13
        b++;                                        // 3.13.1
        c = b * b;                                  // 3.13.2
    }                                               // 3.14
    printf ("The biggest sqrt root of %d is %d", a, b);   // 3.15

    for (a = 0; a < 10; a++) {                      // 3.16
        b++;                                        // 3.16.1

    }                                               // 3.17

    return 0;                                       // 3.18
}                                                   // 4
```

参考文献

[1] GCC5[EB/OL], http://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ

[2] GCC Bugs[EB/OL], https://gcc.gnu.org/bugs/

[3] Bugs – LLVMLinux[EB/OL], http://llvm.linuxfoundation.org/index.php/Bugs

[4] Java Bug Database[EB/OL], http://bugs.java.com/

[5] RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification[S]. Washington D.C.: RTCA Inc., 1992.

[6] RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification[S]. Washington D.C.: RTCA Inc., 2011.

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

[8] McCarthy, John, and James Painter. "Correctness of a compiler for arithmetic expressions." Mathematical aspects of computer science 1 (1967).

[9] Morris, F. Lockwood. "Advice on structuring compilers and proving them correct." Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1973.

[10] Chirica, Laurian M., and David F. Martin. "An approach to compiler correctness." ACM SIGPLAN Notices. Vol. 10. No. 6. ACM, 1975.

[11] Stepney, Susan, et al. "A demonstrably correct compiler." Formal Aspects of Computing 3.1 (1991): 58-101.

[12] Bahr, Patrick, and Graham Hutton. "Calculating correct compilers." Journal of Functional Programming 25 (2015): e14.

[13] Blum, M., and S. Kanna. "Designing programs that check their work." ACM Symposium on Theory of Computing ACM, 1989:269-291.

[14] Pnueli, A., M. Siegel, and E. Singerman. Translation validation. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 1998:151-166.

[15] Glesner, Sabine. "Using Program Checking to Ensure the Correctness of Compiler Implementations." 2003:191--222.

[16] Gaul, Thilo, Antonio Kung, and Jerome Charousset. "AJACS: Applying Java to Automotive Control Systems." Conference Proceedings of Embedded Intelligence. 2001.

[17] COQ DEVELOPMENT TEAM. The Coq proof assistant reference manual[J]. TypiCal Project, 2012.

[18] Leroy, Xavier. "Formal verification of a realistic compiler." Communications of the ACM 52.7 (2009): 107-115.

[19] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." ACM SIGPLAN Notices. Vol. 46. No. 6. ACM, 2011.

[20] Rodr ́guez, Leonardo, Miguel Pagano, and Daniel Fridlender. "Proving Correctness of a Compiler Using Step-indexed Logical Relations." Electronic Notes in Theoretical Computer Science 323 (2016): 197-214.

[21] Krivine, Jean-Louis. "A call-by-name lambda-calculus machine." Higher-Order and Symbolic Computation 20.3 (2007): 199-207.

[22] Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12: 576-580.

[23] Floyd R W. Assigning meanings to programs[C]//Proceedings of Symposium on Applied Mathematics, 1967, 19-31.

[24] Magill S, Nanevski A, Clarke E, et al. Inferring invariants in separation logic for imperative list-processing programs[C]//Proc of the 3rd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2006), Charleston, 2006: 47-60.

[25] Ireland A. Towards automatic assertion refinement for separation logic[C]//Proc of the ASE 2006. [S.l.]: IEEE Computer Society, 2006: 309-312.

[26] Owre S, Rushby J, Shankar N. PVS specification and verification system[J]. URL: pvs. csl. sri. com, 2001.

[27] COQ DEVELOPMENT TEAM. The Coq proof assistant reference manual[J]. TypiCal Project, 2012.

[28] Wenzel M, Paulson L C, Nipkow T. The isabelle framework[M]//Theorem Proving in

Higher Order Logics. Springer Berlin Heidelberg, 2008: 33-38.

[29] Clarke E M, Grumber O, Peled D. Model Checking[M]. Cambridge: MIT Press, 1999.

[30] Baier C, Katoen J P. Principles of Model Checking[M]. Cambridge: MIT Press, 2008.

[31] Li Y, Li Y, Ma Z. Computation tree logic model checking based on possibility measures[J]. Fuzzy Sets and Systems, 2015, 262: 44-59.

[32] Rozier K Y. Linear temporal logic symbolic model checking[J]. Computer Science Review, 2011, 5(2): 163-203.

[33] Stirling C, Walker D. Local model checking in the modal mu-calculus[J]. Theoretical Computer Science, 1991, 89(1): 161-177.

[34] K L McMiIIan. SymboIic modeI checking: an approach to the state expIosion probIem[R]. Carnegie-MeIIon University, Department of Computer Science, Report CMU-CS-92-131.1992.

[35] Holzmann G J. The model checker SPIN[J]. IEEE Transactions on Software Engineering, 1997, 23(5): 279-295.

[36] R CIeaveI, J Parrow, B Steffen. The concurrency workbench: a semantics-based verification tooI for the verification of concurrent systems[J]. ACM Transactions on Programming Languages and Systems, 1993, (51): 36 -72.

[37] McCarthy J, Painter J. Correctness of a compiler for arithmetical expressions[C]//Mathematical Aspects of Computer Science 19: Proc of Symposia in Applied Mathematics, 1967: 33−41.

[38] Pnueli A, Siegel M, Singerman E. Translation Validation[C]//Proc. 4th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems. 1998:151-166.

[39] Fang Yi. Translatin of Optimizing Compilers[D]. New York University, 2005.

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

攻读硕士学位期间取得的学术成果

[1] 陈志伟，谭宇，马殿富. 一种基于安全 C 的编译器形式验证方法，第二十六届全国抗恶劣环境计算机学术年会，2016：375-384.

致谢

行文至此，意味着我的硕士生涯已接近尾声。回忆过去的点点滴滴，其中充满了酸甜苦辣，更有自己在各方面的收获与成长。在此我要感谢陪我度过这几年时光的老师、同学、家人和朋友们。

首先由衷感谢我的导师马殿富教授。在马老师淳淳教诲和耐心指导下，我顺利完成了自己的学位论文。从研究课题的选择、项目进展、论文的修改，直到最终论文的完成，所获得的每一点成果背后都默默凝聚着他的辛劳和汗水，衷心感谢马老师对我的指导和关怀。马老师是一位知识渊博、兢兢业业、桃李芬芳的好老师，他的言传身教不仅引领着我迈进了学术研究的大门，还教会了我很多做人的道理，让我学会了谦虚和规则的内涵，这些受益一生的教诲将时刻感染和激励着我。在此谨向马老师致以衷心的感谢和深深的敬意。

感谢计算机新技术实验室的刘旭东老师、韩军老师、胡春明老师、沃天宇老师、林学练老师、赵永望老师、李建欣老师和张日崇老师等，他们开阔的视野、高深的学术知识和年轻的心态，使整个实验室沐浴在和睦的工作氛围和活跃的研究环境中，为实验室取得一项又一项的科研成果带下了坚实的基础。

感谢谭师兄把我领进编译验证课题的大门，也让我意识到了自己科研和处事上的不足。还要感谢实验室的所有师兄弟姐妹们在平时开展相关工作中互相的支持和帮助。特别感谢我们这一届的同学。

感谢爸爸妈妈对我的养育和支持，虽然外出求学多年以来并不在你们身边，但你们无微不至的照顾和关心我依旧能深深的感受到，也正是因为有了你们的无私奉献，才能让我有勇气走到现在并且在未来的人生中继续披荆斩棘。

感谢陪伴多年的好友——光神、小海和婷姐，在此突然有一种“此地一为别，孤蓬万里征”的感觉，但友谊就如同久酿的老酒，越久越香醇。

最后，我要向百忙之中参与我的毕业论文审阅、评议和答辩的各位老师表示由衷的感谢！