**中图分类号：TP 311.5**

**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 |
| 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 |

**The Research and implementation of proof algorithm of Program Correctness based on B\* specification language**

A Dissertation Submitted for the Degree of Master

**Candidate: Yin Shunshun**

**Supervisor: Prof. Han Jun**

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

机载软件是安全关键软件，对安全性和可靠性有着极高的要求。随着机载软件向高度综合化、模块化方向发展，机载软件系统设计日益复杂，代码量快速增长，现有的民用航空适航验证标准DO-178B无法融入不断涌现的诸如模型开发与验证、面向对象技术和形式化方法等软件开发的新技术。新的民用航空适航验证标准DO-178C对DO-178B进行了补充和修订，明确将形式化方法引入到机载软件的开发和验证过程中。

相较于传统软件开发方法，形式化方法使用严格的数学方法对软件进行开发，并对最终的软件系统进行形式化验证，从理论上保证软件的正确性，更能满足安全关键软件的安全性和可靠性需求。然而，现有形式化验证方法往往验证过程复杂，且对验证者的理论要求较高，对大规模软件的验证代价十分巨大。基于现有形式化验证方法实现的验证工具，虽然能在一定程度上提高形式化验证的效率，但其使用门槛较高，过度依赖人的交互证明，对循环程序的证明依赖于循环不变式的构造。

针对于现有形式化验证方法和形式化验证工具存在的问题，本文提出了一种基于语境的形式化验证方法，该方法以B\*语言为形式规范语言，从程序的形式化描述推导出其形式化功能逻辑表达式，并通过对功能逻辑表达式正确性的验证完成对程序正确性的验证。该方法提出了语境的概念，用于记录程序证明的状态，辅助用户证明，并使用限定数学归纳法证明循环结构程序，避免了循环不变式的构造。本文基于提出的方法，实现了一个基于语境的形式化验证工具，该工具支持对非循环结构程序的自动证明，大大提高了验证的效率；支持对非循环结构程序的交互证明，能够引导用户给出循环的功能逻辑表达式，降低了证明难度。

关键词：机载软件，形式化验证，B\*语言，语境，限定数学归纳法

**Abstract**

Large airborne software, as the safety critical software, has strict requirement on safety and reliability. As the design of airborne software become more and more complex and the code quantity grows rapidly, DO-178B, existing airbone software airworthiness standard, is difficult to guarantee the high requirement of safety and reliability because it is unable to integrate new techniques such as object-oriented technology and formal method. DO-178C, the newer standard to DO-178B, gives supplement revision to DO-178B and explicitly introduces the formal method to the development and verification of airborne software.

Compared to traditional software development methods, formal methods can theoretically guarantee the correctness of software by using rigorous mathematical methods to develop and verify software, better meeting the high safety and reliability requirement of safety critical software. However, existing formal verification methods usually make process complicated for verification, usage difficult for users and cost large for large-scale software. Although formal verification tools based on existing methods can improve verification efficiency to some extent, they have over-reliance on human interaction and high threshold for usage. Besides, existing tools must design loop invariant which is difficult to find for the verification of loop program.

In order to overcome the problems of verification methods and tools, this paper proposes a new formal verification method which is based on context and uses B\* language as formal specification language. Different from existing verification methods, this method inference the functional logic expression from program, and verify the correctness of program by verifying the correctness of the derived functional logic expression. This method proposes the concept of context and uses context to record the status of program proof. More importantly, this method takes limited mathematical induction into the verification of loop program to avoid the design of loop invariant. Based on proposed method, this paper implements a formal verification tool. This tool can verify the non-loop program automatically so that it will greatly improve the verification efficiency. For the loop program, this tool can guide the user to give the functional logic expression of loop code and reduce the verification difficulty.

**Key words**: airborne software, formal verification, B\* language, context, limited mathematical induction

**目 录**

[第一章 绪论 1](#_Toc406690181)

[1.1 研究背景及意义 1](#_Toc406690182)

[1.2 国内外研究现状 3](#_Toc406690183)

[1.2.1 程序正确性证明方法 3](#_Toc406690184)

[1.2.2 形式化方法 4](#_Toc406690185)

[1.2.3 形式化验证工具 4](#_Toc406690186)

[1.3 研究目标和研究内容 5](#_Toc406690187)

[1.4 课题来源 7](#_Toc406690188)

[1.5 论文的组织结构 7](#_Toc406690189)

[第二章 形式化验证方法关键技术研究 8](#_Toc406690190)

[2.1 程序正确性证明方法 8](#_Toc406690191)

[2.1.1 Hoare逻辑 8](#_Toc406690192)

[2.1.2 归纳构造演算 9](#_Toc406690193)

[2.2 形式化方法和软件规范形式语言 10](#_Toc406690194)

[2.2.1 B方法 10](#_Toc406690195)

[2.2.2 B\*形式化方法 11](#_Toc406690196)

[2.3 比较 12](#_Toc406690197)

[第三章 基于语境的形式化证明方法 14](#_Toc406690198)

[3.1 基于语境的形式化证明思路 14](#_Toc406690199)

[3.2 基于语境的形式化证明语义的设计 16](#_Toc406690200)

[3.3 证明策略 17](#_Toc406690201)

[3.4 小结 19](#_Toc406690202)

[第四章 基于语境的形式化证明算法 20](#_Toc406690203)

[4.1 语境算法 20](#_Toc406690204)

[4.2 基于语境的语义规则映射算法的设计 21](#_Toc406690205)

[4.2.1 赋值规则映射算法 22](#_Toc406690206)

[4.2.2 条件选择规则映射算法 23](#_Toc406690207)

[4.2.3 循环规则映射算法 24](#_Toc406690208)

[4.2.4 函数调用规则映射算法 24](#_Toc406690209)

[4.2.5 返回规则映射算法 24](#_Toc406690210)

[4.3 基于语境和限定数学归纳法的循环交互证明算法的设计 25](#_Toc406690211)

[4.4 小结 26](#_Toc406690212)

[第五章 基于语境的形式化验证工具的设计与实现 27](#_Toc406690213)

[5.1 基于语境的形式化验证工具需求分析 27](#_Toc406690214)

[5.2 基于语境的形式化验证工具概要设计 28](#_Toc406690215)

[5.3 基于语境的形式化验证工具详细设计和代码实现 30](#_Toc406690216)

[5.3.1 B\*程序解析模块 30](#_Toc406690217)

[5.3.2 语境管理模块 33](#_Toc406690218)

[5.3.3 语义规则映射模块 35](#_Toc406690219)

[5.3.4 用户交互模块 37](#_Toc406690220)

[5.4 基于语境的形式化验证工具的应用示例 37](#_Toc406690221)

[5.5 小结 39](#_Toc406690222)

[第六章 基于B\*的Arinc653内核建模与验证 41](#_Toc406690223)

[6.1 Arinc653标准 41](#_Toc406690224)

[6.2 Arinc653内核建模 42](#_Toc406690225)

[6.2.1 Arinc653高层抽象机建模 42](#_Toc406690226)

[6.2.2 Arinc653低层抽象机建模 45](#_Toc406690227)

[6.2.3 Arinc653抽象机实现 48](#_Toc406690228)

[6.3 Arinc653内核验证 49](#_Toc406690229)

[6.4 小结 56](#_Toc406690230)

[结论与展望 57](#_Toc406690231)

[工作展望 58](#_Toc406690232)

[附录 59](#_Toc406690233)

[参考文献 65](#_Toc406690234)

[攻读硕士学位期间取得的学术成果 67](#_Toc406690235)

[致谢 68](#_Toc406690236)

**图 目**

[图1 基于语境的形式化验证工具证明流程图 29](#_Toc405155083)

[图2 基于语境的形式化验证工具功能模块图 30](#_Toc405155084)

[图3 加法模拟乘法的B\*程序语法树示例 31](#_Toc405155085)

[图4 B\*语法树结点UML类图 32](#_Toc405155086)

[图5 B\*值类型UML类图 33](#_Toc405155087)

[图6 语义规则映射模块映射规则类图 35](#_Toc405155088)

**表 目**

[表1 Hoare逻辑公里规则及解释 8](#_Toc406689128)

[表2 Hoare逻辑与归纳构造演算比较 13](#_Toc406689129)

[表3 B方法与B\*方法比较 13](#_Toc406689130)

[表4 语境计算算法 20](#_Toc406689131)

[表5 语境计算算法（续） 21](#_Toc406689132)

[表6 程序证明算法Proof\_Program 21](#_Toc406689133)

[表7 程序证明算法Proof\_Program（续） 22](#_Toc406689134)

[表8 赋值语句映射算法Proof\_Assignment 22](#_Toc406689135)

[表9 条件选择语句映射算法Proof\_Selection 23](#_Toc406689136)

[表10 函数调用规则映射算法Proof\_Function\_Call 24](#_Toc406689137)

[表11 返回语句映射算法Proof\_Return 25](#_Toc406689138)

[表12 循环语句映射算法Proof\_While 25](#_Toc406689139)

[表13 循环语句映射算法Proof\_While（续） 26](#_Toc406689140)

[表14 加法模拟乘法程序B\*源代码Multiply\_Use\_Addition 31](#_Toc406689141)

[表15 BNode成员域及成员方法解释 32](#_Toc406689142)

[表16 BNode成员域及成员方法解释（续） 33](#_Toc406689143)

[表17 BStarValue类成员域及成员方法解释 34](#_Toc406689144)

[表18 BStarSet类成员域及成员方法解释 34](#_Toc406689145)

[表19 BStarStruct类成员域及成员方法解释 34](#_Toc406689146)

[表20 ProofStrategy类成员方法解释 35](#_Toc406689147)

[表21 Context类成员方法解释 36](#_Toc406689148)

[表22 AssignmentRule类成员方法解释 36](#_Toc406689149)

[表23 SelectionRule类成员方法解释 36](#_Toc406689150)

[表24 WhileRule类成员方法解释 37](#_Toc406689151)

[表25 FunctionCallRule成员方法解释 37](#_Toc406689152)

[表26 加法模拟乘法的证明序列及功能逻辑表达式 38](#_Toc406689153)

[表27 加法模拟乘法的证明序列及功能逻辑表达式（续） 39](#_Toc406689154)

[表28 进程管理模块数据结构定义 43](#_Toc406689155)

[表29 进程管理模块B\*高层抽象机 43](#_Toc406689156)

[表30 进程管理模块B\*高层抽象机（续1） 44](#_Toc406689157)

[表31 进程管理模块B\*高层抽象机（续2） 45](#_Toc406689158)

[表32 SUSPEND函数需求说明 45](#_Toc406689159)

[表33 SUSPEND函数需求说明（续） 46](#_Toc406689160)

[表34 SUSPEND函数B\*低层抽象机 46](#_Toc406689161)

[表35 SUSPEND函数B\*低层抽象机（续1） 47](#_Toc406689162)

[表 36 SUSPEND函数B\*低层抽象机（续2） 48](#_Toc406689163)

[表37 SUSPEND函数抽象机实现 48](#_Toc406689164)

[表38 SUSPEND函数抽象机实现（续） 49](#_Toc406689165)

[表39 SUSPEND函数形式验证证明序列及功能逻辑表达式 50](#_Toc406689166)

[表40 SUSPEND函数形式验证证明序列及功能逻辑表达式（续1） 51](#_Toc406689167)

[表41 SUSPEND函数形式验证证明序列及功能逻辑表达式（续2） 52](#_Toc406689168)

[表42 SUSPEND函数形式验证证明序列及功能逻辑表达式（续3） 53](#_Toc406689169)

[表43 分区管理 53](#_Toc406689170)

[表44 进程管理 53](#_Toc406689171)

[表45 进程管理（续） 54](#_Toc406689172)

[表46 分区间通信采样端口服务 54](#_Toc406689173)

[表47 分区间通信队列端口服务 54](#_Toc406689174)

[表48 时间管理 54](#_Toc406689175)

[表49 分区内通信缓冲区服务 54](#_Toc406689176)

[表 50 分区内通信黑板服务 54](#_Toc406689177)

[表 51 分区内通信信号量服务 55](#_Toc406689178)

[表 52 分区内通信事件服务 55](#_Toc406689179)

[表 53 健康监控 55](#_Toc406689180)

[表54 ARINC653内核验证与L4内核验证比较 55](#_Toc406689181)

[表55 B\*语言的关键字 59](#_Toc406689182)

[表56 B\*词法规则 59](#_Toc406689183)

[表57 B\*词法规则（续1） 60](#_Toc406689184)

[表58 B\*词法规则（续2） 61](#_Toc406689185)

[表59 高层抽象机语法结构 61](#_Toc406689186)

[表60 高层抽象机语法结构 61](#_Toc406689187)

[表61 抽象机语法结构 61](#_Toc406689188)

[表62 B\*语言包含、声明和定义 61](#_Toc406689189)

[表63 B\*语言包含、声明和定义（续） 62](#_Toc406689190)

[表64 B\*语言包含、声明和定义 62](#_Toc406689191)

[表65 B\*语言表达式文法 63](#_Toc406689192)

[表66 B\*语言表达式文法（续） 64](#_Toc406689193)

[表67 B\*类型 64](#_Toc406689194)

# 绪论

## 研究背景及意义

随着计算机应用技术的快速发展，计算机软件已经在航空航天领域中得到了广泛的应用，现代飞机几乎所有重要功能系统都与机载软件密切相关。同时，机载软件作为符合A/B级[]安全关键系统的重要构成部分，其内部结构越来越复杂、应用环境越来越开放，机载软件的高安全性和高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，包括许多内部组件和算法及其之间复杂的交互，因此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，即GNU Compiler Collection的简称，是一套由GNU开发的编程语言编译器，它表示GNU C Compiler只能处理C语言，现在逐渐扩展为包含了C++、Objective-C++、Java、Fortran和Go语言等，已经成为Linux下最重要的编译工具之一。目前GCC5（2014）已经飙升了1450万行代码[]，单独处理C语言编译的部分代码量也已经达到100万行左右。虽然GCC已经发展得较为成熟，不仅功能非常强大，结构也异常灵活，便携性与跨平台支持也十分出色，但GNU C Compiler本身一直以来都存在许多问题。GCC官方建立了一个网站专门列出了GCC中已知的bugs，并鼓励用户提交bug报告[]，以使这些bugs能在GCC的下个版本中得到修复。其它的商业编译器也依旧存在着各种各样的问题，如LLVM[]、Java编译器[]等。这些编译器中已发现的出于各种原因，如效率等没被修改的bugs和没有被发现的bugs将一直存在，因此由它们编译生成的可执行的目标代码是无法满足机载软件的高安全性和高可靠性要求的。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[]是美国航空无线电委员会（RTCA）于1992年12月发布的航空适航认证标准体系《机载系统和设备认证中的软件要求》标准。DO-178B标准体系规定了软件开发过程中的各阶段软件制品所要达到的安全目标，对机载软件系统的安全性提出了严格的要求，但未规定针对特定的安全认证目标软件开发方所应提供的安全证据以及在软件开发过程中安全证据的技术和方法。RTCA于2012年又发布了DO-178C[]。DO-178C对DO-178B的补充有四个方面：软件工具验证、基于模型的开发和验证、面向对象编程和形式化方法。DO-178C强调了形式化方法在开发和验证软件的有效性和重要性，并明确将形式化方法引入到机载软件的开发和验证过程中去。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。

传统检测编译错误的方法是进行大量的软件测试，但是测试只能证明软件是有错的，不能证明软件是没有错误的。近年来，形式化验证方法在编译器编译过程正确性验证中得到了持续的关注。形式化验证方法基于严格的数学理论，将软件系统和性质都用逻辑方法来规约，通过基于公理和推理规则组成的形式系统，以如同数学中定理证明的方法来对软件系统进行证明。当前，广泛使用的编译器形式验证技术有定理证明、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，用于对一个计算机系统的正确行为属性进行判断。模型检测的基本方法是用一个状态迁移图M 来表示所要检测的系统的模型，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，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目标代码之间的语义等价性的形式化方法，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。MISRA-C本是汽车制造业嵌入式C编码标准，从MISRA-C:2004开始其应用范围扩大到其他高安全性系统。在MISRA-C:2004中，共有强制规则121条，建议规则20条，并删除了15条旧规则。任何符合MISRA-C:2004编程规范的代码都应该严格的遵循121条强制规则的要求，并应该在条件允许的情况下尽可能符合20条建议规则。采用MISRA-C:2004规范也会对程序有负面影响，比如可能会影响代码量、执行效率和程序可读性等，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子集严格要求了编译器的成熟度及稳定性，编译器必须真实地反映源代码的结构和语义，以便编译前后的代码比较和追踪。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，我们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-178B\C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。

## 国内外研究现状

编译器是一个非常复杂的符号转换程序，要对其进行完全、有效且自动的形式化验证依旧存在着巨大的困难，国内外研究者们几十年来针对此问题进行了孜孜不倦的研究并已经取得了显著的进展，有力的推动了整个编译领域和形式化验证理论的不断进步。下面将从编译验证、程序检验和证明工具三个方面来介绍相关领域的现状及所面临的问题。

### 编译验证

编译正确性是研究如何确保编译过程的正确, 使得编译后的目标代码与源代码执行时的行为相同的，即语义是否保持一致。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这个证明过程的思想为后续研究者打开了编译正确性形式验证的大门。

F.Lockwood Morris（1973）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创建的转换示意图对编译器正确性的研究产生重要的影响, 后续许多研究者对都这个图进行了修改或者扩展。

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。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。

Susan Stepney（1991）展现了另一种验证编译正确性的思路，他们描述了如何从一种语言的形式定义中构造编译器，这种编译器因为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。最后，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。把操作语言写成DCTG语法的形式，最终完成成了编译器的构建工作。

PATRICK BAHR和GRAHAM HUTTON（2015）也开发了一种简单但通用的技术，允许通过系统计算的方法从高级语义派生出正确的编译器，所有编译器的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的语言特征及其组合的，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。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的概念，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正确的构建编译器，它不仅指导用户完成计算过程，而且检查其正确性。此外，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。

### 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。检验器本身就是一个应用于程序P的程序C。程序P所运行的任何实例I，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。同时，他们将现代的密码学方法，特别是概率交互证明的思想，应用于群理论计算的程序检查器的设计中。最后，它描述了一些可以检验的问题，如Sorting、Matrix Rank和GCD等。

Amir Pnueli、Michael Siegel和Eli Singerman（1998）[]提出了使用翻译确认的方式来检验编译器的正确性。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性，也方便使用其他工具来验证证明脚本。

2003年，Sabine Glesner[]描述了使用另一种程序检验的方式来确保编译器的正确性。程序检查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。他扩展了检验场景，要求在检验编译器实现的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（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。

### 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公布了CompCert编译器的开发和形式验证进展。CompCert编译器能完成从一种结构化的函数式语言Clight 到汇编代码 PowerPC 的转换, 整个过程由八种不同的中间语言之间的转换构成, 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（包括goto），部署并证明更多优化方法，经语义一致性证明扩展到共享内存的并发中等。Yang X（2011）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，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，没有找到任何错误代码的错误。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等人证明了一个call-by-name类型函数语言的编译器的正确性，并且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lambda演算的扩展，目标语言是Krivine抽象机的扩展。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。他们所有的结果也都使用Coq辅助证明工具进行了形式化和证明。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和应用他们的方法到其它更接近实际汇编代码的执行模型中。

## 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验证方法来保证编译过程的正确性。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系统工具原型。该工具不仅能完成基本的编译功能，如词法分析、语法分析等，还可以检查源代码是否符合安全C标准；能够使用我们提出的语义验证方法对整个编译过程进行验证并正确的生成目标代码；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，即词法分析和语法分析等中加入对安全C约束规则的检验过程，使得不符合安全C标准的源代码在初始阶段就能被识别出，同时需要结合实际实现对安全C子集做出一定的强制规定。

(2) 研究一种基于语义的形式验证方法验证编译过程是否正确。基于形式文法和自动机的相关理论，可以把对于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语义的一致性证明。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，又基于一阶逻辑的公理系统，设计命题自动推理算法，从公理系统中事先给定的公理（如，目标码模式命题）出发，根据推理规则推导出一系列新命题，并作为前提加入到之后的证明过程中。比较最终推导出来的证明序列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) 基于以上的编译验证工具的设计与实现。编译验证工具要能对输入的源代码自动识别出不同的文法单元，对于普通的运算、赋值等语句由于其语义较简单，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，还要使用证明工具保证其语义的一致性。最后生成目标代码并设计一种整个过程的记录方法和工具的界面展示。

## 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”。

## 论文的组织结构

第一章介绍了论文的研究背景，对国内外关于形式化验证的工作进行了总结和分析，详细说明了本文的研究目标和研究内容。

第二章介绍了国内外经典的基于定理证明的程序正确性证明方法和形式化方法，详细介绍了形式化方法中与本文密切相关的关键技术，并对它们的优势和面临的问题做出了详细的分析和总结。

第三章给出了软件需求的形式化描述，详细阐述了基于语境的形式化证明思路，并将该思路与传统形式化证明思路做了比较。然后基于本文的形式化证明思路，提出了基于语境的形式化证明方法，给出了基于语境的形式化证明语义的设计和证明策略。最后对证明策略的正确性给予了证明，保证了证明方法的正确性。

第四章在本文设计的形式化证明语义的基础上，提出了基于语境的语义规则映射算法，详细介绍了基于语境和限定数学归纳法的循环交互证明算法。

第五章介绍了形式化验证工具的设计与实现，并从需求分析到代码实现的软件开发过程中的每个过程进行了详细分析。最后给出两个应用示例来展示证明形态。

第六章阐述了使用本课题组提出的B\*形式化方法及本文提出的基于语境的形式化证明方法对ARINC653系统进行建模及验证的详细过程。

总结部分对全文进行了总结，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。

# 编译形式化验证相关技术

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，包括定理证明、模型检测和程序检验三种形式化技术，并分析了相关技术的优劣性。本部分是本文技术研究和实现的重要参考。

## 定理证明技术

定理证明技术是将软件系统和性质都用逻辑方法来规约，通过基于公理和推理规则组成的形式系统，以如同数学中定理证明的方法来证明软件系统是否具备所期望的关键性质的方法。主要的计算机辅助定理证明工具有PVS、Coq和Isabelle等，这些证明工具表达能力强, 可以很方便地证明程序的正确性。

### 形式推理

形式推理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时可以假定它们成立。这时可以说目标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 ├ 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。换句话说，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，大部分时候人们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，是证明推演时的某些最基本东西的形式化。我们将介绍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：

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：

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：

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 ├ 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 ├ Q的推演式，这时就可以断言，现在已经有了对推演式HYP ├ Q的一个证明。可以形式化地写为：

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├ P的形式，则有：

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，只需证明所有的前提就够了。实际中可以设法找出一条推理规则，其后件与需要证明的推演式S相同，然后将选出的这条规则以反方向方式使用，产生出若干新的需要证明的推演式。再对得到的每个新推演式做同样的事情，并继续下去，直到某个时刻剩下的推演式都是我们推理规则集合里的公理。一旦证明完成，推演式S就被称为是一个公理，最后把S加入规则集合中。可以发现，对一个推演式的证明时相对于一个特定的推理规则集合完成的。

### 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[]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：，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，执行终止且终止时的状态满足Post，则正确性公式为真，程序P具有完全正确性。在霍尔逻辑中存在一组证明规则，称为霍尔规则。这些规则是语法制导的，它们把证明一条复合命令的部分正确性断言简化成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。

在命令式程序验证方面，基于经典逻辑的霍尔逻辑得到了广泛的应用。但是，对使用指针的命令式语言程序进行推理验证是困难的。分离逻辑是对霍尔逻辑的一个扩展，通过提供表达显示分离的逻辑连接词以及相应的推导规则，消除了共享的可能，能够以自然的方式来描述计算过程中内存的属性和相关操作, 从而简化了对指针程序的验证工作。分离逻辑被证明具有更强的验证能力，如对并发程序和资源管理的验证，继霍尔逻辑之后，分离逻辑有望成为程序形式验证的一种重要方法。

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] s h表示整个堆h被分成两个不相交的部分h0和h1，并且对子堆h1断言P成立，而对子堆h1断言 Q 成立。形式化表示如下：



其中h0⊥h1表示堆h0和h1不相交, h0 . h1表示堆h0和h1的联合。

[P\_\*Q] s h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：



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。

### 辅助定理证明工具

目前定理证明比较好的方式是使用编程和证明统一的框架，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[19]是原型验证系统(Prototype Verification System)的缩写。该系统主要包括规约语言和定理证明器两部分，并且还集成了解释器、类型检查器及预定义的规约库和各种工具。PVS提供的规约语言基于高阶逻辑，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的定理证明器以交互方式工作，同时又具备高度的自动化水准。它的命令的能力很强，琐屑的证明细节为证明器的内部推理机制掩盖，使得用户仅在关键决策点上控制证明过程。PVS为计算机科学中严格、高效地应用形式化方法提供自动化的机器支持。

在计算机科学中，Coq[]是一个交互式定理证明器。它提供了一种形式语言来书写数学定义、可执行算法和定理以及用于机器检查证明的半交互式开发的环境。Coq不是自动定理证明器，而是包括自动定理证明策略和各种决策程序，它还允许用户定义自己的证明方法的策略语言。Coq实现了一个程序规范和名为Gallina数学高级语言。Gallina语言基于称为归纳建构的微积分的表达形式语言，其本身结合了高级逻辑和丰富类型的函数编程语言。通过命令的本地语言，Coq允许定义可以有效评估的函数或谓词，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和通过相对较小的认证“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。最后，作为数学形式化或程序开发的平台，Coq为高级符号、隐式内容和其它各种有用的宏也提供了支持。

Isabelle[]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。它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（Forwards Proof）和后向证明（Backwards Proof）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。

## 模型检测技术

模型检测是一种验证给定系统是否满足给定待测属性的形式验证技术。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来表示所要检测的系统的模型，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 |=φ，则系统的正确性得到证实（verified）；否则，就表明系统中存在错误，即M |= ~φ，系统正确性被证伪（falsified）。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[7]、线性时序逻辑[8]和命题μ演算逻辑[9~10]。

目前，模型检验技术已在协议和硬件的设计及检验上取得了成功，但因为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，从制定初始状态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, 生成反例路径证明被验证属性如何被破坏。

状态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。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。

模型检测的优点在于可以完全自动地进行验证，这一方法的成功在很大程度上应归功于有效的软件工具的支持，主要有SMV、SPIN和CWB等。

## 程序检验技术

### 编译过程正确性

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编译前后源代码和目标代码的语义一致性，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，故对编译过程正确性形式化的证明就是证明对应的转换示意图[22]的成立。



Fig.2 Compiler correctness diagram

图中的箭头可看成是函数映射过程。由图中的上下两条转换路线，我们以得到以下等式是成立的：



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：



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，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。转换示意图对编译器正确性的研究产生重要的影响，后续许多研究者对其进行了修改或者扩展，如Burstall和Landin[]等，但是基本原理是一致的。

### 基本思想

程序检验[26~27]是一种用于检验编译器或代码生成器的源和目标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减弱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，从而大大减轻了证明的难度及工作量。同时，由于检验器不关心编译器的具体实现，因此没有限制编译器的设计以及未来的优化完善等，且检验器是可重用的。

### 证明过程

一个自动化的翻译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的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，它允许证明代表着生成的目标代码的一个语义框架的模型，正确的实现了代表着源代码的另一个模型；（4）通过Analyzer执行证明方法的自动化，如果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：



Fig.3 Program checking process

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，它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。该反例包括了其中生成的代码行为不同于源代码的情景。因此，该反例提供的证据表明，编译器有故障，需要加以修改。

## 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X所示。

表 X 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；会影响编译器本身和性能；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；验证简单且工作量第 | 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，辅助定理工具是当前以及未来进行定理证明的主要手段。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可读性好的反例。

模型检测技术是给要检测的系统建立一个用状态状态迁移图来表示的模型，然后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。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，但其会受到无穷状态空间问题的限制。

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。它是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。

# 编译形式化验证与安全C编译器构建的设计

## 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的重要保证。A级软件开发标准要求软件的错误率为10-9以下，DO-178B/C提出了一套符合A级标准的开发方法。DO-178B/C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，只能证明软件有错误，不能证明软件没有错误，这些只经过软件测试的编译器是不可能达到高安全性的编译系统要求的。因此，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，从而得到了一个经过完全的形式验证的编译器，这是一种直接对编译器本身进行的验证的方法；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代码和目标代码的语义是否一致，即编译过程的正确性，而不验证编译器本身等。这些现有的形式验证方法虽然使编译器能达到一定程度上的高安全性，但却也存在着一些问题，如定理证明方法的验证结果虽然安全可靠，但自动化程度较低；程序检验的方法自动化程度高，但建模和验证的过程较困难等。若能合理的避免现有的这些形式化验证方法的短处取其精化，从而设计出一种新的形式验证方法，应能达到编译系统的高安全性需求。

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语义验证方法，这种方法不是直接验证编译器本身，而是和程序检验的思路一样验证源代码和目标代码的语义是否一致。程序检验方法中是对源代码和目标代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，形式验证难度变大。本文提出的语义验证方法却能把源代码和目标代码整体的验证转化为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于语义验证方法来说只是文法单元数量的增加，源代码结构的复杂化对应着文法单元识别算法的复杂性增加，这些都不会影响到每个文法单元的验证。基于命题逻辑公理系统，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。

## 形式验证过程

根据符合安全关键领域规范以及A级软件开发和验证活动的过程和目标，本文提出的基于模型和形式化方法的编译验证系统开发与验证过程必须符合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X所示。



**图X 编译验证系统开发与验证过程**

上图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。高层需求为文法单元的逻辑命题，这是由安全C子集的文法所规定的。低层需求为编译验证结构模型，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的构建这两方面。软件架构为自动机的并集，也可以说是上下文无关文法的并集，因为文法和自动机存在着映射关系。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代码的自动机实现代码和编译验证代码集成到系统中，这是一种软件集成过程。本文主要研究的是编译形式验证方法和安全C编译系统名的构建两方面，并基于上述DO-178C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。

## 编译形式化验证方法

### 命题逻辑系统

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。涉及命题的逻辑领域称为命题演算或命题逻辑，它描述和研究语句相互结合形成新语句的方式，即逻辑的语法部分。

复合命题的新命题是已知的原子命题用逻辑联结词组合而来。逻辑联结词主要包括∨“或”（析取）、∧“且”（合取）、¬“非”（否定）、→“蕴涵”（条件式）、↔“当且仅当”（双条件式）。命题逻辑的BNF范式如下表X所示。

表 X 命题逻辑的BNF范式

|  |
| --- |
|  |

下面给出命题逻辑系统中一些重要定义。

**定义3.1** 真值赋值（truth valudation）是一个函数，它赋给每个命题唯一的真值，使得它在每个复合命题上的真值取值与某个真值表相一致。因此。比如当且仅当，而当且仅当或。我们成使为真，如果。

**定义3.2** 令为一个（可能无穷）命题集合。我们称是的**后承**（记作），如果对任意的赋值，

。

在上述定义中称为的**前件**。注意，如果为空集，（或写成）当且仅当 永真。

**定义3.3** 令为一个（可能无穷的）命题集合。我们称赋值是的模型，如果对所有的成立。用来表示所有模型的集合。

根据上述基本概念，下面给出命题逻辑中证明系统的形式化定义。其中，在此系统下表示可证性。

**定义3.4** 令为一个命题集合。

（i）一个**从出发的证明**（proof from ）是指一个由构成的有穷序列，使得对每个，以下三条之一成立：

（1）是中的元素；

（2）是公理；

（3）可以通过应用推理规则，由之前的某些推出。

（ii）是从**出发可证的**（provable from ），记作，如果存在一个从出发的证明，这里。

（iii）的**证明**（proof）仅仅是指从出发的证明；是可证的（provable），如果从出发，它是可证的。

命题逻辑系统的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。推理规则必须满足一下条件：如果我们可以从永真命题出发，利用推理规则推出，那么也是永真的。下面分别出给命题逻辑的公理、定理和推理规则。

**公理**3.1 命题逻辑的系统公理由如下形式的命题组成：

（i）

（ii）

（iii）

这里，和可以是任意命题。

上述命题形式通常称做公理模式（axiom scheme）。公理即是上述模式的所有实例，因为、和可以取遍所有的命题。下面给出与本系统相关的定理表述（证明过程略去）。

**定理**3.1 从前件出发的**可靠性**和**完全性**

是从命题集合****出发可证的，当且仅当是****的后承，即。

**推论**3.1 **可靠性**和**完全性**

命题是可证的，当且仅当它是永真的，即。

下表X给出命题逻辑系统的推理规则，其中，HYP为前提或者断言，P、Q为命题。

表X 命题逻辑系统推理规则

|  |  |  |
| --- | --- | --- |
| 名称 | 前件 | 后件 |
| **Rule1** CI规则（合取引入） |  |  |
| **Rule2** 合取删除规则 |  |  |
| **Rule3** 演绎规则 |  |  |
| **Rule4** MP规则（假言推理） |  |  |
| **Rule5** 归谬规则1 |  |  |
| **Rule6**归谬规则2 |  |  |

注意：Rule5和Rule6都被称为“归谬规则”，是同一个规则的两种表达形式。

本文提出的形式化验证方法是基于上面描述的命题逻辑系统的公理和推理规则，对由目标码模式映射成的目标码模式命题集****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****中得到，重复上述过程直到不在变化为止。最终得到的命题集经过专用推理规则整理后即为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，推导出的结果正好符合了按照DO-178C规定的编译验证系统开发与验证过程的高层需求。

### 形式语义

形式语义学是研究形式语言及程序的语义的学问。语义涉及到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。形式语义学的基本方法是用一种元语言将程序加工数据的过程及其结果形式化，从而定义程序的语义。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，这十分有有助于编译程序的设计。一般的程序设计语言的语义主要分为四种：操作语义、指称语义、公里语义和代数语义。与本文的形式验证方法相关的语义只有两种，即操作语义和指称语义，下面将简单介绍一下。

操作语义通过规定程序设计语言在抽象机器上的执行过程来描述程序设计语言的含义。程序设计语言操作语义的表达是通过给出能确定该语言的任何程序的执行效果的一种机制，这样的机制是一个“抽象自动机”：一种能够形式地执行程序的形式设备。操作语义的刻画是通过描述语言的各种构造在解释自动机上的执行效果来实现的。操作语义给出了语言的具体和直观的描述，并且这种描述接近于实际程序。操作语义的缺点在于不是独立于实现。

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。指称不是实在的事物，一般是完全偏序集（CPO）上的高阶函数，这些函数将程序映射成代表其实际运行结果的抽象实体。指称语义分为直接指称语义和接续指称语义。指称语义固有的、区别于其他语义形式的性质为：它是可合成的，即任何表达式含义由其子表达式含义决定。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表示形式。

指称语义和操作语义间具有一定联系，完全抽象的指称语义与操作语义是完全等价的。所以，针对操作语义难以推理的情况可以转化为指称语义的推理来进行。本文的形式化验证方法中使用的语义是指称语义，使用的目标代码是基于Power PC指令集。如下表X给出and指令的操作语义和指称语义。

表X and指令的操作语义和指称语义

|  |  |
| --- | --- |
| 指令用法 | and rA, rS, rB (Rc=0)  and. rA, rS, rB (Rc=1) |
| 操作语义 | result← (rS) & (rB)  rA ← result  if Rc=1 then do  LT ← result< 0  GT ← result> 0  EQ ← result = 0  CR[0] ← LT || GT || EQ || SO |
| 指称语义 | GPR[rS]' = GPR[rA]&GPR[rB]  Rc=1 → CR[0]'={GPR[rA]&GPR[rB] < 0, GPR[rA]&GPR[rB] > 0, GPR[rA]&GPR[rB] = 0, XER.SO} |

由上表可以看出指称语义与操作语义在本质上的等价性。指令的操作语义是Power PC 官方文档直接给出的。那如何得到指令的指称语义呢？表面上看可以简单地从操作语义使用Verilog语言转化得到，但本质上涉及到了CPU的建模和形式验证技术十分复杂。CPU的建模和形式验证技术是由本课题组的其他同学研究完成，本文直接使用了他们的研究成果，在此表示感谢。下表X给出了部分Power PC汇编指令的指称语义。

表X Power PC汇编指令的指称语义（部分）

|  |  |  |
| --- | --- | --- |
| 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] |

上表中，GPR（General-Purpose Register）表示Power PC的通用寄存器，主要用作堆栈指针、函数的第一个参数和返回值等。CR（Conditional Register）为条件寄存器，可以反映某些操作的结果（比如cmp指令），协助测试和分支转移指令的执行。MEM代表内存空间，存储了局部、全局等变量的值。@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在前人的工作中已经进行过形式验证，因此，本文中将把它们直接作为专用公理加入到编译的验证过程中。

### 形式验证方法

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正确性进行形式化验证。

设源代码为S，编译过程用compile表示，目标代码为T，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：

 (1).

又因为目标代码是由源代码编译得到的，故有：

 (2)

成立。由(1)、(2)式可以得到：

 (3).

不妨，再令C文法单元为，目标码模式为。根据上下文无关文法的独立性，则有：

 (4).

把(4)式代入(3)式有：





 (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。对符合安全C子集规定的文法单元来说，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。对这三类文法的验证虽然整体思想一样，但具体实现中却还是存在着一些差异的。其它的C语言中规定的文法，如include语句、宏定义、变量声明等，由于它们的结构过于简单，所以只要确定它们符合C语言定义的语法结构就可以断定是正确的，我们将不会对它们进行形式验证。下图X将给出本文提出的形式验证方法对全C子集规定的文法单元的验证流程。



**图X 形式验证流程**

编译形式化验证方法的几个重点说明：

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。由于证明序列中的每一项都是前提、公理或者定理，又因为命题逻辑公理系统是可靠且完全的，所以证明序列中的每一项一定是正确的，从而最终推导出来证明序列一定是正确的。

命题映射算法：将目标码模式按照专用公理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。算法将遍历目标码模式中的每条指令并用指令对应的指称语义来替换，替换加入到新的命题集合中，即可得到目标码模式的命题。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，因为可以简化对整个验证的过程的描述。

自动推理算法：是本文的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，利用命题逻辑系统的推理规则和公理对命题集进行推理，推理过程还包括了对推理策略进行遍历搜索的过程。把得到的新命题加入到原命题集中，并依据情况去掉多余的旧命题，直到最终的命题集不在变化为止。

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归纳法的形式，也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。

### 验证方法架构

编译验证方法主要有两种方式：一种是验证编译器本身，即验证编译器的实现没有任何问题，不会篡改源代码的语义；另一种是验证编译过程，即验证编译产生的目标代码语义和源代码语义是一致的。很明显，编译器自身的正确是前提, 编译过程的正确是最终的目的。验证编译器是通过验证编译器自身没有任何错误来证明它不会对编译过程产生任何影响，间接的验证了编译过程的正确性。验证编译过程的正确性是本文的目标，下面将给出编译正确性验证的整体架构，如图X所示。



**图X 编译正确性验证架构**

编译过程正确性验证的对象是源代码和目标代码，但直接验证二者的语义是否一致太过于困难，所以本文提出了基于文法单元和目标码模式的编译语义验证方法。从上图可知，首先需要根据程序设计语言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，用一种类似于树型的数据结构维持不同文法单元之间的上下文关系。然后，每当自动机识别出一个文法单元后时会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，所以比较固定。接着，对目标码模式使用形式验证方法进行验证，即可得到每个文法单元和生成的目标代码的语义是否一致。最后，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，从生成目标代码。

从编译正确性验证的整体架构中可以看到，验证方法的本质其实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X所示。



**图X 文法单元验证架构**

文法单元验证的主要目的是验证每个文法单元和对应目标码模式的语义是否一致。从上图可知，当程序识别出文法单元i并生成对应的目标码模式i后，首先需要使用专用公理把每个目标码模式映射为对应的目标码模式命题，因为目标码模式在本文中特指的是Power PC汇编代码片段，无法直接进行推理。这一映射过程由后面将介绍的命题映射算法实现，其中的专用公理正是前文所述的Power PC指令的指称语义。然后，在形式推理模块内基于输入的目标码模式命题使用命题逻辑系统的推理规则和公理对其进行推理，当目标码模式命题的规模不在变化时，结束整个形式推理过程。接着，对最终的目标码模式命题进行取语义操作后（），得到得到目标码模式的语义。最后，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，即可完成整个证明过程。若语义一致性得到保持，则输出每个文法单元在形式推理中的证明序列，方便其它工具对证明过程的检验。

## 安全C编译器构建方法

### 编译技术说明

编译器是一个庞大、复杂的软件系统，其本质是一个语言翻译程序，负责把高级语义程序翻译成低级汇编语言程序。低级汇编语言程序是不能直接被计算机执行的，必须依靠汇编器把它翻译为计算机硬件可识别的二进制语言程序。本文中我们只讨论从高级语言程序到低级的汇编语言程序这一翻译过程，而汇编代码到二进制代码的转换是一种简单的一一对应关系，我们将不会涉及。下面将给出现代编译器的典型三阶段结构，如图X所示。



**图X 编译器结构**

现代编译器在结构上可划分为前端、中间部分（优化器）以及后端三个阶段。每一阶段都有一组不同的问题需要解决，用于解决这些问题的方法也各有不同。前端的工作涉及理解程序并将其分析结果以IR（中间表示）的形式记录下来，包括词法分析、语法分析和加工三个过程；中间部分专注于该进IR的形式，一般包括一个或多个优化技术的使用；后端必须将转换过的IR程序映射到目标机器的有限资源上，从而能够有效的利用资源，这一阶段包括目标机器上的指令选择、调度和寄存器分配等过程。编译器的三个阶段共享了同一个基础设施。

在这三个阶段中，中间部分是最错综复杂的。优化器以IR程序作为输入，产生一个语义上等价的IR程序作为其输出，它的目的是为了改进IR程序。优化器的实现涉及到了许多经典的编译优化算法，如常量传播、冗余消除、复写传播和死代码消除等算法，这些算法会被设计为若干趟规模较小的处理过程。鉴于编译优化部分不会影响IR程序的语义且作者水平所限，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。

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展开、文替换和注释删除等。预编译器在工业上的标准实现类似于编译器实现的内容，它也必须分析源代码的语义才能进行预处理过程。本文中为了简化整个编译构建过程，我们把预编译阶段放到了编译器的前端实现中，一般在词法分析阶段即可同时完成整个预处理过程。

（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、语法等处理过程，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、语法、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，而是在下一阶段遍历上一阶段的产出物。

（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面所述的安全C子集，它是MISRA-C:2004的子集且结合了航天型号软件的特点。MISRA-C:2004基于的标准是C90，即ISO 9899:1990 [2] 定义的 C 编程语言。总之，本文的高级语言规范本质中基于的是C程序设计语言。

（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。另外，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。如前文所述，本文不涉及。

### 编译构建方法

本文编译部分的构建主要分为前端和后端两个阶段。前端专注于理解并分析源程序，后端主要将程序映射到目标机器的指令集和有限的资源上。为了支持本文提出的编译形式验证方法，在前端的实现上必须支持对文法单元的处理等一系列操作。传统的编译实现技术中并不存在文法单元的概念，所以需要对编译的传统实现技术进行修改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，而对语法分析部分需要设计新的处理算法重新实现，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，经过文法单元的形式验证后，即可生成最终的目标代码。下面给出本文中编译构建方法的架构，如图X所示。



**图X 编译系统架构**

上图可分为上下两部分，上部分是编译系统的构成部分，下部分为输入的数据在整个处理流程中的转化过程。其中，编译系统的构成分为前端和后端两部分，前端中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。形式验证正是前文所述的文法单元的验证过程，本文中将把它作为一个处理阶段加入到编译系统的后端实现中。

编译器的前端必须将源程序的结构与程序设计语言的定义进行比较，以对其进行检测。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范式表示法来描述。这是一种格式上有点改变和缩短了的上下文无关文法，大大简化了程序设计语言的定义和编译程序的结构。下面将给出与后文讨论到的技术相关的部分文法，如下表X所示。完整的安全C子集文法见附录I。

表 X 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>  ... |

表中，文法的描述被分为了两个部分，即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。文法的词法规则部分由词法分析器来实现，主要的任务是将字符流变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，其中个各分量的含义如下：

（1）是有限状态集合，以及一个错误状态。

（2）是一个有限字母表。

（3） 是一种状态转移函数。它将每个状态和每个字符的组合映射到下一个状态。在状态遇到输入字符，FA将采取转移。

（4）是指定的起始状态。

（5）是接受状态，。

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：表示FA从起始状态处理输入字符所发生的状态转移。产生的状态接下来作为输入，该状态连同又输入到产生下一个状态；依次类推，直至耗尽所有的输入。最后一次应用的结果仍然是一个状态。如果该状态是，那么串就被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转移到错误状态，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回调式下推自动机。回调式下推自动机是一个六元组，其中：

（1）是状态的一个非空有穷集合。

（2）是输入字母表。

（3）是栈字母表。

（4）是起始状态。

（5）是栈起始符号。

（6）是终止状态集，。

（7）是状态映射函数。它表示从当前状态出发，且栈顶符号是z，若输入的字符为c，则映射到三元偶。其中，为跳转到的下一个状态，为跳转状态执行结束后返回的状态，为动作结束后的栈顶元素。

基于上述回调式下推自动机，文法单元的识别过程为：表示下推自动机从起始状态出发且单词栈顶元素为，处理输入单词所发生的状态转移，用一个三元组表示，把入单词栈。然后，以作为初始状态进行FA的转移，不断重复FA的转移过程，直到到达FA的终结状态，就完成了一个文法单元的识别，把FA最终的状态写入返回。最后，从出发不断重复上述过程，直到处理完所有的输入单词流。如果最终状态是F，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。

### 编译其它方法

层级编码是对源代码按照代码的嵌套层次进行编码并把所得的编号映射到编译出的目标代码及相应的证明序列中的方法。主要意义在于实现了从源代码到目标代码级和相应证明序列的互相追踪，完成了DO178B/C规定的A级软件可追踪性需求。层级编码方法是在编译的前端实现的，因为其只用到了少量的语法分析的信息。主要的处理过程为：在预处理阶段使用栈的方式存储当前的层级编号，若当前行是复合语句的入口，则把编号1压入栈中；若当前行是复合语句出口，则弹出栈顶元素；若在复合语句内，只是行数加一，则把当前的栈顶元素加一。编号的结果需要保留下来传递给后续的处理阶段，同时还要把对源代码编码结果写回源代码中。

安全C子集规定了本文编译验证系统处理高级语言的规范，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的要求。安全C检验是为了分析源代码是否符合安全C子集的规范，它是对直接在编译器的构建中使用安全C子集进行设计和约束的补充。安全C子集中有一些规则，如不允许使用goto语句、continue语句等，可以直接在编译验证系统的构建中实现；但对于另一些规则，如不支持递归函数和动态堆内存分配等，就需要我们设计单独来实现其检验算法了。这些检验算法的实现部分分布的较为分散，并不能使用一个直接的检验器集中完成检验过程，而是需要把它们和编译前端的词法分析、文法单元识别等阶段结合起来。安全C子集中还有一些关于文档的规则，如字符集和相应的编码应该文档化等，这些规则需要我们在程序之外编写文档来实现。

## 小结

本章主要论述了编译形式化验证方法和安全C编译器构建方法的设计，以及验证过程与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到系统中。

（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每个C文法单元的验证。采用形式推理技术，在命题逻辑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的等价性是否一致，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，采用了有限自动机和回调式下推自动机作为编译前端的主体实现算法，解决了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DO178B/C规定的A级编译验证系统的可追踪性和高安全性需求。

# 编译形式化验证与安全C编译器构建的关键技术

## 编译形式化验证关键技术

### 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，几乎所有的程序设计语言都是通过上下文无关文法来定义的。上下文无关文法之间具有相互独立的特性，从而上下文无关文法的并集仍然是上下文无关文法。安全C子集也是由多种上下文无关文法描述的，每种上下文无关文法产生的语句结构都可以由对应的自动机来识别，这种语句结构的C语言表达形式就是C文法单元。

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，本文引入了语境的概念，根据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X给出了部分C文法单元和其对应的语义。

表 X C文法单元和语义

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

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

### 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具体语境下的并集。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X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 X 目标码模式

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

程序的形式化验证需要特定的公理系统作为基础。命题逻辑公理系统是从一些被当做公理的命题（前提）出发，根据演绎法，推出一系列定理的演绎体系。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。下表X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和命题。

表 X 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>  cmpi 7,0,0,0  bne 7,.L1 | P1 = .L1:  P2 = <STA-LIST>  P3 = GPR[0] = <LOG-EXP>  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 |

### 推理证明

本文提出的证明方法是基于命题逻辑的公理系统，从公理系统中事先给定的公理（目标码模式命题）出发，根据命题逻辑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。由于证明序列中的每一项都是前提、公理或者定理，又因为命题逻辑的公理系统是可靠的，所以证明序列中的每一项一定是正确的，从而最终推导出来证明序列一定是正确的。

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X所示。

表 X <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 ∧  .L1: ∧  <STA-LIST\_2> ∧  .L2:  } | S5, S6, S7, S8, S9, S10, CI |
| S12= {  (<LOG-EXP> < 0 -> <STA-LIST\_1>) || (<LOG-EXP> > 0 -> <STA-LIST\_1>) || (<LOG-EXP> == 0 -> <STA-LIST\_2>)  } | S11, REDUCE |
| S13= (<LOG-EXP> != 0 -> σ(<STA-LIST\_1>) || <LOG-EXP> == 0 -> σ(<STA-LIST\_2>)) | S12, σ |

表X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>)，

结合表X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X所示。

表 X <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 + @.L1 || <LOG-EXP> == 0 -> PC = PC + 4 | S7,S8,MP |
| 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 |

由上表可知，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，得到的目标码模式的语义将无法含有循环的语义，这是由于命题逻辑系统无法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)⋀(∀n)(P(n)→P(s(n))→(∀n)P(n)，限定数学归纳法是在一般数学归纳法的基础上，限定n是有穷的，即对于循环结构程序循环是可终止的，终止条件由人给出。下面将结合<while-statement>文法单元的语义，对表X中S10公式使用限定数学归纳法来证明，如下表X所示。

表 X <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。 |

## 编译形式化验证算法

### 命题映射算法

命题映射算法的作用是把目标码模式转换为命题的形式，以方便进行后续的推理证明。算法中首先需要把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，对于目标码模式中的每一条目标码把其指令名和参数分离开，并去掉无关的符号。接着通过判断指令名和参数构成数组的长度来决定指令的类型，若数组长度为0时，则是空行，不进行后续的处理；若数组长度为1时，则可以直接把目标码加入命题集；若数组长度大于1时，需要用指令相应的指称语义来替换目标码后，才能把新的命题加入命题集。最后，完成上述目标码模式的遍历后，即可把目标码模式的指称语义表示成命题集的形式输出。下面将给出命题映射算法的伪代码，如表X所示。

表 X 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** |

由上表可知，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。注意这里我们假设了算法中其它函数的处理时间为一个常数。

### 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使用命题逻辑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，我们还会对命题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X所示。

表 X 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 != 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** |

由上表可知，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(n2)，效率有待提升。

### 循环交互证明算法

对于不包含循环的目标码模式命题集可以直接对推导出的证明序列进行取值（σ）操作，从而得到目标码模式命题的语义，与C文法单元的语义比较，可以直接判断出二者的语义是否保持一致。对于包含循环的目标码模式命题集，直接对推导出的证明序列进行取值操作却无法完成语义一致性检验过程，需要引入循环交互证明算法才能完成整个验证过程。

循环交互证明算法的理论基础是限定数学归纳法。算法首先引导用户输入n为1时C文法单元的语义，然后按照循环条件分别为真或假时，分别构造新的命题加入到4.2.1命题映射算法输出的命题集的一个copy中，调用4.2.2中的自动推理算法对copy命题集进行推理，从而得到此时目标码模式命题的语义。

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

表 X 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** |

由上表可知，循环交互证明算法其实只需要执行两遍即可，第一遍是初始阶段证明n=1时，待证结论是否成立；第二遍假设第n=N时待证结论成立，推导n=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(n2)。

## 安全C编译器构建关键技术

### 词法分析方法

词法分析是编译处理流程中很重要的一步，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，那么如何从输入的字符流中分析出每个词法记号呢？为了解决这个问题，需要引入有限自动机。

有限自动机能解析并识别出词法记号，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的词法记号。有限自动机分为两种，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，故本文将使用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 X 标识符有限自动机

上图中，标识符的识别从S0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。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，并且第1个字符必须为字母或下划线。



图 X 整数有限自动机

整数的有限自动机如上图所示，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S0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X所示。

表 X 有限自动机识别算法

|  |
| --- |
| **Algorithm 4 Finite** **Automata Recognition Algorithm** |
| **Input:** CharStream, S, SA  **Output:** Token |
| 1: char ← NextChar()  2: state ← *S0*  3: **while** char ≠ *eof* and state ≠ *Se* **do**  4: add char to token  5: state ← NextState(state, char)  6: char ← NextChar()  7: **end while**  8: **if**  **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，SA为接受状态集合，*Se*代表一个错误的状态，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，则下一个状态就错误状态*Se*。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SA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，给出错误提示即可。

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，这里使用了自动机的并操作。下图X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 X 自动机的并操作

词法分析中可以在起始状态读入一个字符时就可以知道这个字符可能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。上图中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，这样就完成了一个词法记号的识别。

### 文法单元识别方法

文法单元识别是从输入的单词流中构造出文法单元的过程，这一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适配到程序设计语言的语法模型中从而得到具有一定语义的语句单元。在4.1.1小节中，我们给出了安全C子集中的部分文法单元，仔细分析可以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。本文引入了有限自动机来进行文法单元的识别，下面将首先给出<if-statement>文法单元，如下表X所示。

**表 X <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。下面给出识别<if-statement>文法单元的有限自动机，如图X所示。

 图 X <if-statement>文法单元有限自动机

上图中，两个状态之间转移的条件是单词，故“(”、“)”、“{”和“}”均不是字符而是词法符号。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S0开始的，若遇到if单词时，就进入S1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S2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S3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S2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同的，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，因而不能直接使用并操作来组合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，下面给出回调式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X所示。

表 X 回调式下推自动机算法

|  |
| --- |
| **Algorithm 5 Callback 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, )  6: push token to AnalysStack  7: push statesStack to RollbackStack  8: **while** RollbackStack not empty **do**  9: pop RollbackStack to statesStack  10: **while** statesStack not empty **do**  11: pop statesStatck to state  12:  **if** **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 token from AnalysStack  35: **end if**  36: **end while**  37: **end while** |

上述算法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的集合，在自动机匹配失败后可以回溯到上一个状态，状态栈中存储当前阶段自动机可能达到的状态。

算法以新建分析栈和回溯栈开始，主体为一个遍历输入的单词流的循环。首先读入一个单词并把当前状态设置为*S0*，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的状态集合。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。再次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。重复上述过程，再次调用NextState函数并把相应的单词和状态入栈。最后，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父状态输入的单词也是无效的，故需要从分析栈中取出栈顶的元素。

### 层级编码方法

本文提出的层级编码方法是构造符合A级软件标准的编译验证系统的重要保证。实践中由于输入的源文件一般包含多个子文件，而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X所示。

表 X 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 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 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并向编号栈放入元素1开始。首先，以行为单位遍历输入的源代码，若代码行中含有‘}’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使层级减一，同时使弹出元素后的编号栈栈顶元素加一。然后，若当前输入的代码行不为空，则交给solveLine函数来识别当前行中的单词，返回solveLine函数识别出的当前中可能存在的所有文件名并加入文件名列。接着，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，若当前输入的代码行不为空，则使编号栈栈顶元素加一。最后，需要遍历文件名列表并把每个引用的文件交给solveMultipleFile函数来处理，solveMultipleFile会递归调用层级编码算法对每个文件进行处理。

## 小结

本章提出了基于语境的形式化证明算法，具体如下：

1. 对基于语境的形式化证明语义中对赋值语句、条件选择语句、循环语句、函数调用语句和返回语句的语义转换规则进行了算法设计。
2. 证明策略的算法设计也被蕴含在各规则映射算法中。证明策略的Read规则将语句结点按照对应的语义转换规则转化为命题（证明项）；Split规则的设计蕴含在条件选择规则映射算法和循环规则映射算法中，自动将条件选择语句的证明分成两个分支；Combine规则会在程序含有多重条件选择语句或多重循环语句时自动使用。
3. 提出了基于语境和限定数学归纳法的循环交互证明算法，将循环结构的功能表达式映射成自然数相关的功能表达式，并通过语境引导用户使用限定数学归纳法给出循环的功能逻辑表达式。

# 编译验证工具的实现

## 基于语境的形式化验证工具需求分析

基于研究目标，基于语境的形式化验证工具应该满足以下三方面要求：

正确性：在证明理论正确性的条件下尽量保证证明工具实现的正确性，并且能在工具验证出错的情况下，可以认为检查出证明序列的错误。

高效性：对于非循环结构程序，应该尽量减少人的参与；对于循环结构程序，应该尽可能的引导用户给出循环功能逻辑表达式。

易用性：应该尽可能降低工具使用门槛，增加用户熟悉度，并为用户提供友好的交互环境，引导用户证明。

对于正确性需求来说，本文提出的基于语境的形式化证明方法是基于公理系统和集合论的，语义转换规则满足语义的正确性，证明策略中的推导规则的正确性第三章已经证明，从而保证了本文验证工具的理论基础是正确的。因为由赋值语句、条件选择语句、循环语句、函数调用语句和返回语句转换为一阶逻辑表达式的转换规则确定，所以证明序列的顺序基本确定，并且每一步的推导都是使用证明策略中的规则，可以很方便地人工检查证明序列的正确性，从而可以在工具实现出错的情况下，发现证明序列的错误。

对于高效性需求来说，本文提出的基于语境的形式化证明方法对于非循环结构程序证明，基本无需人的参与，只需要用户给定待验证的B\*源程序作为输入，并且人工判断工具推导出的程序功能逻辑表达式的正确性，避免了传统验证方法中需要人工逐句构造中间断言的工作量，大大提高了验证效率；对于循环程序的证明，工具会自动给出循环前几步的证明项以及语境，引导用户发现循环规律，给出假设。工具会在假设的基础上继续自动证明，帮助用户使用数学归纳法验证假设的正确性。避免了传统验证方法中循环不变式构造困难的问题，降低了证明难度的同时，也提高了证明的效率。

对于易用性需求来说，首先，工具的输入是B\*语言书写的程序。B\*语言在语法上与C语言类似，只保留了基本数据类型（如整型、浮点型）、元组类型（类似于C语言中的结构体）和集合类型，工具使用者书写时会比较熟悉，降低了工具的使用门槛；其次，基于语境的形式化证明方法是从直接从程序推导出其功能逻辑表达式，推导出的证明序列基本确定，便于人工检查推导的正确性；最后，语境概念的引入，可以十分方便的帮助用户发现变量间关系，了解当前程序的证明状态，进一步简化了交互证明难度，提高了工具的易用性。

针对于以上要求和分析，本文设计和实现的基于语境的形式化验证工具应该满足以下需求：能对B\*书写的程序进行词法分析，语法分析并生成B\*语法树，并能在词法或者语法分析出错时为用户提供出错信息，即实现一个B\*的编译器；支持语境的概念，方便用户了解程序证明状态和查询变量间关系；支持基于语境的形式化证明算法，满足对非循环结构程序的自动证明；提供持久化服务，将程序的证明序列以及功能逻辑表达式保存到磁盘上；尽可能减少用户交互；尽可能减小交互难度；保证算法实现和证明策略实现的正确性。

上述需求中，前4条为本文实现的验证工具的功能性需求，后3条为非功能性需求。

## 基于语境的形式化验证工具概要设计

基于语境的形式化验证工具的证明流程图如图1所示。验证工具的输入是B\*程序，然后经过B\*程序解析，如果语法分析和词法分析正确，则生成对应的B\*语法树，否则输出错误信息，验证结束。在得到B\*语法树之后，根据证明语义中的并置规则，按顺序依次抽取语法树中的语句结点进行证明，并判断语句结点的类型来选择相应的规则映射算法和证明策略，直到最后一条语句结点被证明完毕，保存证明序列，保存程序的功能逻辑表达式，验证结束。

本文提出的验证工具的框架设计如图2所示。验证工具主要由B\*程序解析模块、语境管理模块、语义规则映射模块、和用户交互模块4个模块组成。

1. B\*程序解析模块：接收用户输入的B\*源程序，对其进行词法分析和语法分析。如果源程序没有错误，则生成相应的B\*语法树，否则为用户输出错误信息。
2. 语境管理模块：记录整个验证过程的上下文及变量状态，为待验证程序语句提供相关语境。当程序语句类型为赋值语句时，需要对被赋值变量的语境进行更新；当程序语句类型为条件选择语句时，对条件进行语境运算，判断条件真伪（如果当前条件可判断），决定验证分支；当语句类型为循环语句时，对循环条件进行语境运算，并且通过给出循环语句相关变量间的关系，引导用户证明。

**图1 基于语境的形式化验证工具证明流程图**

1. 语义规则映射模块：程序验证的理论依据，根据语句结点类型和当前语境，映射到相应的语义规则映射算法，利用证明策略中的Read规则，将语句结点转换为一阶逻辑表达式。对于条件选择语句结点和循环语句结点，在使用Read规则后需使用Split规则将其转化为两个分支去证明；对于含多重条件选择语句和多重循环语句的程序证明，需在每一步使用Combine规则将所有前提合取作为新前提。
2. 用户交互模块：为用户输出证明项，帮助用户了解当前验证进度。遇到循环结构程序时，为用户输出循环语句前几步的验证结果，引导用户输入循环的功能逻辑表达式和循环终止条件，并接收用户输入，从而完成循环的验证。

**图2 基于语境的形式化验证工具功能模块图**

## 基于语境的形式化验证工具详细设计和代码实现

### B\*程序解析模块

给定用户输入的B\*源程序，为满足验证的输入要求，需对其进行词法分析和语法分析将其转换成相应的程序语法树作为验证的输入。因此，程序解析模块是整个工具的开始。

基于上文给定的B\*语言的词法和文法，本文使用一个开源的语法解析器javacc进行程序解析。javacc（Java Compiler Compiler）是一个用[JAVA](http://zh.wikipedia.org/wiki/JAVA)语言开发的开源的语法解析器，支持跨平台运行，能够读取与上下文无关且有特殊意义的语法文字，并将这些语法文字转换成可以识别的并且能够与该语法匹配的编译器程序，而且还可以使用解析器自带的JJTree工具帮助用户创建程序语法树。

在本文实现的工具中，javacc接收B\*的BNF范式，生成B\*的编译器程序，对用户输入的B\*程序进行词法分析和语法分析，并利用JJTree生成B\*的语法树作为整个验证过程的输入。

如表14给出的使用加法模拟乘法的B\*源代码，其使用javacc进行词法分析、语法分析后生成的语法树如图3所示。

表14 加法模拟乘法程序B\*源代码Multiply\_Use\_Addition

|  |
| --- |
| **FunctionName: Multiply\_Use\_Addition**  **Input:** multiplier, multiplicand // 乘数和被乘数，multiplicand为正数  **Output:** product //返回乘积 |
| 1: **Def**: **int Multiply\_Use\_Addition** (int multiplier, int multiplicand)  2:{  3: **int** result = 0;  4:  **int** multiplicandRecord = multiplicand;  5: **while**(multiplicandRecord != 0)  6: {  7: result = result + multiplier;  8: multiplicandRecord = multiplicandRecord – 1;  9: }  10: **return** result  11:}  12: **End** |

语法的根节点是Machine结点，类似于B语言，Machine被称为抽象机。Operations结点代表抽象机中的操作，在本文中可以理解为函数。FunctionDefine结点是函数定义结点，Operations结点的子孙由FunctionDefine节点组成。FunctionDefine有5个子结点，分别是NormalType结点（返回值类型结点）、IdNode结点（函数名）、ParaDefineList结点（函数形参结点）、Attributes结点（函数局部变量定义结点）和Statements结点（函数体语句结点）。所有的叶节点表示抽象机的终结符，中间节点为非终结符。



图3 加法模拟乘法的B\*程序语法树示例

本章使用UML类图来表示模块代码的设计。在UML类图中，每个类的矩形方框第一栏代表类的名称，第二栏代表类的属性（成员域），第三栏代表类的成员方法。-号代表属性或者方法是private（私有的），+号代表属性或者方法是public(公有的)，#代表属性或者方法是protected（受保护的）。栏中每一行冒号前边的标识符是属性名或者方法名，冒号后边的标识符代表属性的类型或者方法的返回类型。带三角形箭头的实指向线代表类之间的继承关系，箭头指向的类为父类，箭头尾端为子类。虚线表示类之间的依赖关系。

B\*语法树结点的UML类图如图4所示，BNode是所有语法树结点（B\*中的终结结点和非终结结点）类的基类，所有结点类都要继承BNode并实现其构造方法。本文给出的UML类图中只是给出了关键的各类型语句结点与BNode的继承关系。



图4 B\*语法树结点UML类图

BNode的成员域和成员方法定义如表15所示：

表15 BNode成员域及成员方法解释

|  |  |
| --- | --- |
| **成员** | **解释** |
| parent | 表示语法树结点的父结点 |
| childern | 表示语法树结点的所有子孙结点 |
| gSymTab | 表示语法树结点能访问的全局符号表 |
| lSymTab | 表示语法树结点能访问的局部符号表 |
| type | 表示语法树结点的值类型（基本类型、集合、元组等） |
| BNode() | 表示语法树结点的构造方法 |

表16 BNode成员域及成员方法解释（续）

|  |  |
| --- | --- |
| jjtAddParent() | 表示向语法树结点的子孙结点中添加结点的方法，用于条件选择语句证明时进行语法树拼接 |
| jjtAccept() | 表示深度遍历当前语法树结点及其子孙结点的方法，并使用设计模式中的观察者模式为遍历时对于不同结点的不同操作留下接口。 |
| getCode() | 表示获取语法树结点对应B\*源代码的方法 |

### 语境管理模块

语境管理模块为程序证明的每一步提供语境获取、语境更新和语境计算的服务。该模块维护了一个使用哈希表结构实现的语境表。一个变量（或者常量）由其标识符和作用域唯一确定，因此语境表的Key由变量名（常量名）和作用域组成，Value存储变量（常量）类型和变量（常量）值。

不同类型的变量其值的结构也不同，因此需要不同的数据结构去模拟。本文关于值类型类的UML类图如图5所示。

**图5 B\*值类型UML类图**

BStarValue: 所有值类型的基类，也是基本类型（如整形、浮点型、字符串等）值的实现类。其成员变量和方法如表17所示：

表17 BStarValue类成员域及成员方法解释

|  |  |
| --- | --- |
| **成员** | **解释** |
| value | 存储变量或常量的真实值，如果真实值不可计算，设为空 |
| isKnown | 表示变量或常量真实值是否能够计算的标志位 |
| changeTime | 记录语境变化，即变量语境改变的次数 |
| getExpValue() | 表示得到语境表达式值的方法，表达式值表现为变量间的关系 |

BStarSet: 集合的值类型的实现类，有集合运算的方法和生成集合表达式值的方法。其成员变量和方法如表18所示：

表18 BStarSet类成员域及成员方法解释

|  |  |
| --- | --- |
| **成员** | **解释** |
| type | 存储变量或常量的真实值，如果真实值不可计算，设为空 |
| BStarSet() | BStar类的构造方法，构造一个type类型的哈希集合（HashSet）去模拟集合 |
| caculate() | BStar类的静态方法，用于集合间二元运算（如交、并、差等）。如果参与运算的两个集合中任意一个集合的真实值是未知的，则运算结果的真实值也是未知的，表示为集合间的运算关系 |
| generateSetExp() | 表示生成集合表达式值的方法，表现为对集合的操作和运算，形如{Aelement1, Aelement2 …} SetB |

BStarStruct: 元组的值类型的实现类，类似于C语言中的结构体，其成员变量和主要方法说明如表19所示：

表19 BStarStruct类成员域及成员方法解释

|  |  |
| --- | --- |
| **成员** | **解释** |
| nameResDimen | 存储元组中各维度变量名或常量名到维度的映射，使用哈希表实现 |
| type | 表示元组中各维度变量或常量类型 |
| BStarStruct() | BStarStruct类的构造方法，构造一个type类型的HashMap去模拟元组。元组每一维可以是任意的类型，也可以是元组 |
| getDesignDimValue | 获取BStarStruct中某一维的值 |
| generateStructExp() | 生成元组的表达式值，表现为对元组各维度的操作和运算 |

BStarPointer: 指针的值类型的实现类，其与基类BStarValue主要的不同在于其需要一个成员变量pointToVariable来标识指针指向的变量。

### 语义规则映射模块

语义规则映射模块由非循环规则映射模块（赋值规则映射、条件选择规则应设、函数调用规则映射和返回规则映射）、循环规则映射模块、证明项生成模块和证明策略模块四个子模块组成，它是整个证明工具的核心模块。该模块的UML类图如图6所示，其涵盖了各个类之间的依赖关系和继承关系。



图6 语义规则映射模块映射规则类图

ProofStrategy是证明策略模块，实现了本文验证工具中的证明策略。它是各规则映射模块的依赖模块，其成员方法说明如表20所示：

表20 ProofStrategy类成员方法解释

|  |  |
| --- | --- |
| **成员方法** | **解释** |
| sequenceRule() | 对应于语义转换规则中的并置规则。因为该规则既不会改变程序证明状态，也不会生成证明项，仅仅是为顺序证明程序提供理论基础，因此将其放在证明策略模块中作为其他各模块的依赖基础 |
| useRead() | 对应于证明策略中的Read规则。它接收语法树语句结点，根据结点类型使用相应的映射规则，生成语义证明项。证明项也使用BNode类型进行保存 |
| useMP() | 对应于证明策略中的MP规则 |
| useSplit() | 对应于证明策略中的Split规则 |
| useCombine() | 对应于证明策略中的Combine规则 |

Context模块是语境管理模块，也是语义规则映射模块的依赖模块。其部分成员方法说明如表21所示：

表21 Context类成员方法解释

|  |  |
| --- | --- |
| **成员** | **解释** |
| caculateExp() | 语境计算方法，用于计算给定表达式的值。主要用于赋值映射规则模块中右表达式值的计算和条件选择和循环规则映射模块中条件真值的计算 |
| getAllChanged() | 该方法是得到所有改变的全局变量和函数参数的语境，使用HashMap保存结果，主要用于返回规则映射模块。一个函数的作用在于改变了全局变量值、参数值和获取其返回值。在函数返回时，应将改变的全局变量和参数也作为程序功能逻辑表达式输出 |
| contextRecover() | 恢复语境表到一个指定的状态，是条件选择规则和循环规则映射模块的依赖方法。条件选择规则映射算法中将会对条件成立和条件不成立（如果条件真值不可计算）两个分支进行验证，需要在验证其中一条分支前对语境进行备份，并在验证之后恢复语境，再验证另外一条分支。循环规则与条件选择规则类似 |
| updateContext() | 更新语境方法，是赋值规则映射模块的依赖算法，用赋值语句右边表达式的值更新被赋值变量的语境 |

AssignmentRule模块是赋值规则映射模块，其主要函数说明如表22所示：

表22 AssignmentRule类成员方法解释

|  |  |
| --- | --- |
| **成员方法** | **解释** |
| getRightExp() | 解析赋值语句结点，获取其右表达式结点 |
| getLeftVar() | 解析赋值语句结点，获取被赋值变量。变量是BStarVariable类的实例，由变量名和作用域唯一确定 |

AssignmentRule模块将调用语境管理模块中的语境计算函数计算右值表达式值，再调用其语境更新函数记录语境的改变。

SelectionRule模块是条件选择规则映射模块，其主要函数说明如表23所示：

表23 SelectionRule类成员方法解释

|  |  |
| --- | --- |
| **成员方法** | **解释** |
| getCondNode() | 解析条件选择语句结点，获取条件结点，用于真值计算 |
| getIfsNode() | 获取If条件成立的需执行的语句结点 |
| getElsesNode() | 获取If条件不成立的需执行的语句结点（Else中的语句结点） |
| combineNodes() | 拼接语句结点的函数，其实现是调用BNode类中的jjtAddChild方法 |

WhileRule模块对应于循环规则映射模块，该模块依赖于MutualProof模块，即用户交互证明模块。MutualProof的主要方法说明如表24所示：

表24 WhileRule类成员方法解释

|  |  |
| --- | --- |
| **成员方法** | **解释** |
| readInput() | 接收用户输入的循环功能逻辑表达式和终止条件，并将其转化为语法树 |
| generateTempProofItem() | 为用户输出循环的前几步证明项，引导用户证明。这些证明项不属于程序最终的证明序列 |
| tempUpdateContext() | 在得到户输入转换成的语法树后，对语法树进行解析，临时更新语境表（如果循环的功能表达式中含有赋值），并在用户输入的基础上，给出循环第N+1步的证明项，供用户检查其假设正确性 |

FunctionCallRule模块对应于函数调用规则映射模块，该模块依赖于ProofStrategy模块中的sequenceRule()方法从而可以将函数体中的语句展开依次证明。其方法说明如表25所示：

表25 FunctionCallRule成员方法解释

|  |  |
| --- | --- |
| **成员方法** | **解释** |
| getFunctionName() | 解析函数调用语句结点，获取函数名 |
| replacePara() | 将被调函数中的所有语句中的形参使用实参进行替换，用于程序证明 |

ReturnRule模块对应于返回规则映射模块，其主要功能是生成程序功能逻辑表达式，记录程序对全局变量和参数语境的改变。

### 用户交互模块

该模块是工具的用户界面，主要有以下两个功能：

1. 用于循环证明时的用户交互，接收用户输入的循环功能逻辑表达式和循环终止条件。
2. 为用户输出程序每一步的证明项和相关语境。
3. 为用户输出程序最终的功能逻辑表达式。

## 基于语境的形式化验证工具的应用示例

本节将通过一个循环的例子展示基于语境的验证工具的验证形态。待验证的B\*源程序如表13所示，源程序的功能使用加法运算模拟乘法运算，其中有循环结构语句。经过本文实现的工具验证，其证明序列和功能逻辑表达式如表14所示。

在验证表14中1-4行代码时，验证工具将multiplier, multiplicand, result和multiplicandRecord记录到语境表中并初始化其语境。当验证至第5行代码时，程序验证遇循环结构，见表26第2行，A1 = w(multiplicandRecord'0 = multiplicand'0){while(multiplicandRecord != 0)}，A代表证明步，A后边的数字代表第几步，w代表语境，w后括号中为当前语句相关的语境。花括号中的为当前代码语句。表26中3-14行根据用户需求给出循环证明（假设循环初始条件成立，即循环至少执行一次）的前两步证明，17-20行、25-27行和29行为用户的交互证明。21-24行对应于循环语句映射算法Proof\_While中的步骤3，步骤4和步骤5目前为人为扩展和确认过程，证明示例中未展示。当用户在29行给出循环最终的功能逻辑表达式后，循环证明结束，继续证明循环后的语句，即Return语句。表26第30行对Return语句进行了验证，函数返回值为乘数与被乘数的乘积，至此，循环初始条件成立的分支验证完毕；第32~33行对循环的另一分支（循环初始条件不成立，循环体不执行）进行验证。两条分支验证完毕后，提取Return语句处的证明步，构成程序的最终功能逻辑表达式，即程序功能项，如图2第35、36行所示。由功能项可以很直观地看出程序的功能至此，该程序的验证过程结束。

表26 加法模拟乘法的证明序列及功能逻辑表达式

|  |
| --- |
| 1: 证明序列:  2: A1 = w(multiplicandRecord'0 = multiplicand'0){while(multiplicandRecord != 0)}  3: 是否继续自动推理？(Y / N)  4: Y  5: A Temp = multiplicand'0 != 0 -> w(result'0 = 0; multiplier'0 = multiplier'0){result = result + multiplier}  6: A Temp = multiplicand'0 != 0 -> result'1 = multiplier'0  7: A Temp = multiplicand'0 != 0 -> w(multiplicandRecord'0 = multiplicand'0){multiplicandRecord = multiplicandRecord - 1}  8: A Temp = multiplicand'0 != 0 -> multiplicandRecord'1 = multiplicand'0 - 1  9: 是否继续自动推理？(Y / N)  10: Y  11: A Temp = multiplicand'0 - 1 != 0 -> w(result'1 = multiplier'0; multiplier'0 = multiplier'0){result = result + multiplier}  12: A Temp = multiplicand'0 - 1 != 0 -> result’2 = multiplier'0 + multiplier'0  13: A Temp = multiplicand'0 - 1 != 0 -> w(multiplicandRecord'1 = multiplicand'0 –  1){multiplicandRecord = multiplicandRecord - 1}  14: A Temp = multiplicand'0 - 1 != 0 -> multiplicandRecord'2 = multiplicand'0 - 2  15: 是否继续自动推理？(Y / N)  16: N |

表27 加法模拟乘法的证明序列及功能逻辑表达式（续）

|  |
| --- |
| 17: 请输入该循环归纳表达式：  18: multiplicand'0 - N != 0 -> result'N+1 = (N+1) \* multiplier'0  19: multiplicand'0 - N != 0 -> multiplicandRecord'N+1 = multiplicand'0 - (N + 1)  20:  21: A Temp = multiplicand'0 - N - 1!= 0 -> w(result'N+1 = (N+1) \* multiplier'0; multiplier'0 = multiplier'0){result = result + multiplier}  22: A Temp = multiplicand'0 - N - 1!= 0 -> result'N+2 = (N+1) \* multiplier'0 + multiplier'0  23: A Temp = multiplicand'0 - N - 1!= 0 -> w(multiplicandRecord'N+1 = multiplicand'0 - (N + 1)){multiplicandRecord = multiplicandRecord - 1}  24: A Temp = multiplicand'0 - N - 1!= 0 -> multiplicandRecord'N+2 = multiplicand'0 - (N + 1) - 1  25: 请输入循环终止条件和最终归纳表达式：  26: 终止条件：N = multiplicand'0  27:  28: A Temp = multiplicand'0 - (multiplicand'0 - 1)!= 0 -> result'multiplier'0 = multiplier'0 \* multiplicand'0  29: 最终归纳表达式：multiplicand'0 != 0 -> w(result'multiplier'0 = multiplier'0 \* multiplicand'0){return result}  30: A2 = multiplicand'0 != 0 -> w(result'multiplier'0 = multiplier'0 \* multiplicand'0){return result}  31: A3 = multiplicand'0 != 0 -> MultiplyUseAddition = multiplier'0 \* multiplicand'0  32: A4 = ~(multiplicand'0 != 0) -> w(result'0 = 0){return result}  33: A5 = ~(multiplicand'0 != 0) -> MultiplyUseAddition = 0  34: 功能项是:  35: A3 = multiplicand != 0 -> MultiplyUseAddition = multiplier \* multiplicand  36: A5 = ~(multiplicand != 0) -> MultiplyUseAddition = 0 |

## 小结

本章首先提出了基于语境的形式化验证工具的三个要求，即正确性、高效性和易用性，并对各个要求的合理性进行了分析。基于提出的要求，明确了验证工具的功能需求和非功能需求。之后给出了以功能模块图和程序流程图的形式给出了工具的概要设计，明确工具应由B\*程序解析模块、语境管理模块、语义规则映射模块和用户交互模块四部分组成，并且分析了各模块的作用和相互关系。然后在概要设计的基础上，以UML类图的形式给出了各个模块的详细设计和代码实现，对各个模块的具体实现和依赖关系进行了阐述，并对其中重要的成员变量、方法和设计模式进行了详细地说明。最后本章给出了一个循环证明的例子来展示工具。

相较于传统的形式化验证工具，本章实现的基于语境的形式化验证工具有以下优势：

1. 针对于传统形式化验证工具使用门槛高的问题，基于语境的形式化验证工具以B\*语言为验证语言。B\*语言在语法上与C类似，并且只保留了基础数据类型、集合和元组，用户容易掌握，从语言的方面降低了使用门槛；并且，基于语境的形式化验证工具只使用证明策略中的Read、MP、Split及Combine规则，并且对于不同程序结构证明策略的使用基本确定，无需用户具备较深的理论基础，从用户交互证明的方面降低了使用门槛。
2. 针对于传统形式化验证工具证明过程不直观的问题，基于语境的形式化验证工具直接从程序推导出其功能逻辑表达式，因为由赋值语句、条件选择语句、循环语句、函数调用语句和返回语句转换为一阶逻辑表达式的转换规则确定，所以证明序列的顺序基本确定，并且语境概念的引入，使得用户可以更好地了解程序证明的状态。因此整个证明过程清晰易懂。
3. 针对于传统形式化验证工具面临的依赖人机交互、交互难度大的问题，基于语境的形式化验证工具能够对非循环结构程序自动推导证明，只需用户确定推出的功能逻辑表达式与事先定义的程序形式规约是否一致，大大减少了人机交互；对于费循环结构程序，基于语境的形式化验证工具能在很大程度上引导用户使用限定数学归纳法进行推导证明，避免了循环不变式的构造，降低了人机交互难度，也一定程度地减少了人机交互。

# 总结与展望

随着民用航空客机机载软件的系统机构日益复杂和代码量的迅速增长，传统的软件开发方法已经逐渐不能满足其对软件安全性和可靠性的极高需求。新的民用航空适航验证标准DO-178C为机载软件的开发引入了形式化方法，建议使用形式化方法对大型机载软件进行形式开发和形式验证。在此背景下，本文阐述了形式化开发方法相对于传统软件开发方法的优势，介绍了形式化验证在保证软件正确性上的重要性。同时，也指出了形式化方法在开发效率和验证效率上所面临的挑战。本文介绍了课题组针对于形式化方法开发效率存在问题而提出的B\*形式化方法，并将其与B方法进行对比，说明了B\*方法的优势。本文针对形式验证效率上存在问题，提出了一阶逻辑和集合论为理论基础的基于语境的形式化证明方法，并基于此方法实现了一个以B\*语言为软件规范形式语言的形式化验证工具，能够高效地对程序正确性进行证明，并且降低了证明难度。

本论文的主要工作包括以下4点：

1. 基于定理证明的形式化验证方法研究。本文从安全关键机载软件的高安全性和可靠性难以保证出发，分析了使用形式化方法对机载软件进行验证的意义，也提出了形式化验证在难度和效率上的问题以及存在问题的原因，指出了提高验证效率和降低验证难度的重要价值。在分析的基础上，提出了本文的研究目标和研究内容，并对与本文密切相关的典型证明方法和规范语言进行了研究分析，为本论文的研究奠定了研究基础。
2. 基于语境的形式化证明方法的设计。本文提出了一种新的形式化证明思路，该思路直接从软件的形式化描述（本文中指B\*源程序）推出其功能形式化描述，并将语境的概念和限定数学归纳法引入到本文提出的证明方法中。基于此种思路，本文设计了一套类似于Hoare逻辑的形式化证明语义规则，规定了程序结构转换为一阶逻辑表达式的转换规则，分析了此方法能提高证明效率以及降低证明难度的原因，并对证明方法的理论正确性进行了证明，为本论文的研究奠定了理论基础。
3. 基于语境的形式化证明算法的设计。本文基于提出的形式化证明方法，为语义规则和证明策略设计了相应的规则映射算法，为本文的形式化验证工具的设计与实现奠定了算法基础。
4. 基于语境的形式化验证工具的实现。本文基于B\*语言和提出的基于语境的形式化证明算法，设计并实现了一个基于语境的形式化验证工具，并对验证工具各模块的设计原则、模块功能以及具体实现做了详细的描述。该工具能够对B\*书写的程序进行词法分析、语法分析，并能够生成B\*的语法树，用于形式验证；能够完成非循环结构程序的自动推导证明，提高了证明的效率；对于循环结构程序，该工具可以引导用户交互证明，无需用户构造循环不变式，降低了证明的难度。

本文的主要贡献包括以下三点：

1. 针对于传统形式化方法中存在的中间断言构造问题，本文提出的基于语境的形式化证明方法从理论上避免了中间断言的构造，大大提高了程序验证效率。
2. 针对于传统形式化方法中循环程序证明困难、依赖于循环不变式的问题，本文提出的基于限定数学归纳法的循环交互证明算法能够避免循环不变式的构造，通过与语境结合，帮助用户发现变量间关系及了解程序证明状态，引导用户使用数学归纳法完成循环程序的证明，从一定程度上降低了程序验证难度。
3. 本文实现的基于语境的形式化验证工具已对工业界认可的Arinc653标准中的49个内核函数（总代码量约为4300行）进行了验证，证明项约为4200行，整个证明使用了约5人月。说明工具已具备万行规模系统的形式化验证能力；并且经过与L4内核验证的对比，说明工具在验证效率上有着一定的优势，具有真正的使用价值。

## 工作展望

未来的工作可以考虑如下三个方面：

1. 在循环交互证明过程中，需要人工判定计算机推出的关于循环第N+1次循环的功能逻辑表达式与用户输入的功能逻辑表达式是否一致，未来可以借鉴主范式相等的思想去自动判定两个逻辑表达式的等价性。
2. 目前Arinc653规范的B\*抽象机描写依旧不是非常精细和完善，有一些内核函数抽象机经过形式验证和代码转化后，不能直接应用于实际中，未来需要进一步完善课题组提出的B\*形式化方法，使抽象机经过形式验证和代码转换后得到的C代码能直接应用于实际中。
3. 验证工具的交互功能需要进一步改善，未来可以为用户提供查询变量间关系、证明依据提供等功能，方便用户了解程序证明状态。

# 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> ') |

# 参考文献

1. Mecham, M. "Boeing Faces Pretty Tight 787 Delivery Schedule." Aviation Week 9 (2007).
2. RTCA Inc., "RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification", Washington D.C.: RTCA Inc., 1992
3. 蔡喁，郑征，蔡开元，王泽新，欧旭坡．机载软件适航标准DO-178B/C研究[M]. 上海交通大学出版社，2013:1-38
4. RTCA Inc., "RTCA/DO-178C: Software Considerations in Airborne Systems and Equipment Certification", Washington D.C.: RTCA Inc., 2011
5. R. W. Butler (2001-08-06). "What is Formal Methods?". Retrieved 2006-11-16.
6. Steve Schneider. The B-Method:An Introduction [M]. Hampshire :PAL GRAVE ,2001.
7. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1977: 238-252.
8. Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.
9. Colón M A, Sankaranarayanan S, Sipma H B. Linear invariant generation using non-linear constraint solving[C]. Computer Aided Verification. Springer Berlin Heidelberg, 2003: 420-432.
10. Dijkstra E W, et al. A discipline of programming[M]. Englewood Cliffs: prentice-hall, 1976.
11. Clarke E M, Grumberg O, Peled D A. Model checking[M]. MIT press, 1999.
12. Abrial J R, Lee M K O, Neilson D S, et al. The B-method[C] VDM'91 Formal Software Development Methods. Springer Berlin Heidelberg, 1991: 398-405.
13. Steve Schneider.The B-Method:An Introduction [M]. Hampshire :PAL GRAVE ,2001.
14. Jones C B. Systematic software development using VDM[M]. Prentice Hall Englewood Cliffs, 1990.
15. Yves Bertot, Pierre Casteran. 交互式定理证明与程序开发—Coq归纳构造演算的艺术[M]. 清华大学出版社，2010.
16. Barthe G, Forest J, Pichardie D, et al. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant[M]. Functional and Logic Programming. Springer Berlin Heidelberg, 2006: 114-129.
17. Gérard Berry and Jean-Jacques Lévy. SCADE: Industrial Success of a Synchronous Language and its Future Challenges. GGJJ 2011, Gérardmer Feb. 3rd 2011. Amar Bouali. Esterel Technologies.
18. Jean-Louis Camus, Bernard Dion. Efficient Development of Airborne Software with SCADE SuiteTM. Esterel Technologies 2003.
19. Paulson, Lawrence C., and Markus Wenzel. "Isabelle/HOL: a proof assistant for higher-order logic." Vol. 2283. Springer, 2002.
20. 维基百科.构造演算[EB/OL], <http://zh.wikipedia.org/w/index.php?title>=构造演算
21. 黄达明, 曾庆凯. 基于分离逻辑的程序验证技术[J]. Journal of Software, 2009, 20(8): 2051-2061.
22. 尹宝林, 何自强等. 离散数学[M]. 高等教育出版社, 1998:50-153.
23. 麦忠凡, 呂卫锋. 程序语言设计原理[M]. 北京航空航天大学出版社, 2011.
24. J. Delange, L. Pautet, Fabrice Kordon. Modeling and Validation of ARINC653 Architectures[J]. In the Embedded Real Time Software and Systems Proceedings(ERTS²), May 2010.
25. Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface part 1 – REQUIRED SERVICES. Aeronautical Radio, 2005.

# 攻读硕士学位期间取得的学术成果

1. 尹顺顺，马殿富，赵永望，赵宪琦. 一种基于语境的程序正确性验证算法的研究与实现，第二十四届全国抗恶劣环境计算机学术年会，2014：297-304.
2. Shunshun Yin, Jun Han, Kuldeep Kumar and Yu Huang. Dependency-Topic-Affects-Sentiment-LDA Model for Sentiment Analysis. The 26th International Conference on Tools with Artificial Intelligence, 2014: 413-418.

# 致谢

在毕业论文完成之际，首先我要感谢我的恩师韩军教授。在研究生期间，韩老师教导我要勇于接受新鲜事物，敢于攻克技术难题。在韩老师的指导下，我阅读了大量的顶尖的机器学习的论文，并对自然语言处理产生了浓厚的兴趣。如果不是韩老师的鼓励，我可能早已在机器学习中复杂的公式推导面前止步不前。韩老师不仅给我了学术上的巨大帮助，更是教会了我很多做人做事的道理，坚持不懈，踏踏实实，才是成功的捷径。韩老师的言行和教诲时刻影响着我，是并一直是我未来人生道路中一盏指明灯。

同时，我要向实验室的马殿富老师致以崇高的敬意。马老师带领我进入了形式化验证这一新鲜的领域。因为我的离散基础不是很好，每次与马老师讨论问题，马老师总会给我细心讲解形式验证领域一些比较难懂的学术问题。马老师对学术的追求，对工作的负责态度，对社会的高度负责感和对学生的关心，给我留下了深深的影响。在此，我衷心地向马老师表达崇高的敬意和感谢。

感谢赵永望老师研究生期间对我的指导和监督。赵老师经常与我交流，细心为我解决学术上遇到的难题。赵老师对工作一丝不苟的态度也深深影响着我。

感谢实验室的怀进鹏老师、刘旭东老师、李欢老师、胡春明老师、沃天宇老师、李建欣老师、孙海龙老师、马帅老师，能在ACT实验室读研，是我莫大的荣幸。

感谢我们课题组的博士赵宪琦师兄，杨宏伟师兄，曾浩师兄。他们在实验室起了很好的带头作用，营造了良好的学术氛围，让我进步很快。

感谢我的师兄胡尹、于晨辉和赵城成，他们给我做了很好的榜样，并给我学术上的指导和生活上的帮助。感谢黄昱，他是韩老师的另一个学生。黄昱对我Java语言的学习帮助很大，并且经常与我讨论学术问题，研究难题，是我研究生期间的挚友。感谢实验室的王磊，吴文娟，王琦，招啟杰，邰振赢同学，谢谢你们研究生的陪伴，让我度过了一个难忘的时光。

特别感谢我的父母，感谢你们多我的关心和鼓励，使我能向着更好的未来前行。

最后感谢我的女友贺妍艳，感谢你陪我走过的日子，虽然你我不在一地，但是你无微不至的关怀让我十分感动，感谢你的付出。