**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

定理证明是形式化验证方法里的一种，目前在学术界的方式是使用编程和证明统一的工具，如Coq、Isabelle和PVS等，下面将主要介绍一下Coq证明工具。

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方法，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。

## 程序正确性证明方法

### Hoare逻辑

Hoare于1969年提出了Hoare逻辑，它为严格地使用数理逻辑证明计算机程序的正确性提供了一套逻辑规则[8]。Hoare逻辑使用Hoare三元组“{P}C{Q}”完成对程序的正确性证明。其中，P和Q是断言，分别被称为前置条件和后置条件，P和Q一起被称为程序的形式规约。C是程序。Hoare三元组的意义是指如果条件P在程序C执行之前是成立的，则Q在C执行完之后也成立。如果C中含有循环结构程序，则C的终止性需被单独证明。Hoare逻辑以三元组形式为基础，为不同的程序结构定义了相应的推导规则，具体规则和解释如表1所示：

表1 Hoare逻辑公里规则及解释

|  |  |  |
| --- | --- | --- |
| 规则名称 | 规则表示 | 规则解释 |
| 空语句公理 |  | 如果前置条件P在执行空语句之前是成立的，那么在空语句执行完之后P也是成立的。 |
| 赋值公理 |  | 表示把P中变量x的所有自由出现都替换成表达式E得到的谓词。 |
| 并置规则 |  | {P} S {Q} 和 {Q} T {R}同时成立能推出{P}S;T{R}也成立，这为顺序证明程序提供了理论基础。 |
| 条件规则 |  | 表示如果执行S，那么执行前P和B为真，如果执行T，那么执行前Ｐ为真，B为假。 |
| 循环规则 |  | 表示在执行S之前I和C为真，S执行完后，I为真C为假，I是循环部分的循环不变式。 |
| 推论规则 |  | 表示如果一个Hoare三元组为真，那么把前置条件加强或者后置条件减弱之后依旧为真。 |

Hoare逻辑是经典的基于定理证明的程序正确性证明方法，是很多现有程序正确性证明方法的基础，如分离逻辑等，已经取得了广泛的应用。然而，在使用Hoare逻辑证明程序时，需要构造每一条语句正确的后置条件，即中间断言，增加了证明的工作量，降低了证明的效率，制约了证明的规模；并且对于循环结构程序的证明，Hoare逻辑依赖于循环不变式的构造，加大了证明的难度。

### 归纳构造演算

Christine Mohring和Thierry Coquand于1988年对构造演算[20]进行扩展，设计了归纳构造演算，将归纳数据类型上的算法的公理化的一些良好性质融入到构造演算中。归纳构造演算使用类型以及类型的示例给出程序的定义，并使用类型系统进行程序证明。归纳构造演算中将类型的类型称为大类sort，并有如下定义。

规范说明（Specification）：Set是归纳构造演算预定义的大类之一，一个类型为Set的项被称为一个规范说明。

程序（Programs）：程序是指一个类型为规范说明的项。

因为类型也是项，所以类型也有类型，类型规则的形式化描述如下：

在归纳构造演算中，利用了递归的思想。Set大类是一个项，因此Set也必然有一个类型，而类型本身也是一个项，故归纳构造演算有一个无穷递归的大类层次结构，这个层次结构称为类型空间。类型空间由Type(i)类型组成，并满足如下关系：

归纳构造演算中的项可使用如下方法构造：

1. T是项（也叫类型）
2. P是项（也叫做命题，是所有命题的类型）
3. 若A和B是项，则(AB)、、也是项。

归纳构造演算有证明、命题、谓词、大类型和T本文5种对象类型，且满足上述形式化描述。

归纳构造演算有如下推理规则：

目前归纳构造演算已成为形式化验证工具Coq的理论基础，并随Coq得到了广泛的应用。但是归纳构造演算基于lambda表达式，在理解上不太直观，且在程序的证明上需要预先定义程序满足的性质的逻辑描述，影响证明的效率和证明的规模。

## 形式化方法和软件规范形式语言

### B方法

法国科学家的J-R Abrial提出的B方法[12,13]是一种基于一阶逻辑和集合论的形式化方法，它从软件的需求说明书出发，定义抽象模型，使得设计者可以不必过早地关心系统实现的细节。B方法通过对抽象模型的逐步精化，并对开发过程中产生的证明义务进行证明，最终得到可执行的系统代码。

抽象机是B方法中使用的基本单元，它主要由数据描述、操作描述和不变式组成。

数据描述指常量和变量，不变式指这些常量和变量要满足的关系（一直满足）。数据描述和不变式一起组成了抽象机的静态行为。操作描述是指抽象机中对数据的操作，是抽象机的动态行为，它可以改变变量的值，但不能违反不变式所制约的数据间关系。在B方法开发过程中，软件系统的抽象模型由抽象机组成。

B方法的开发过程一般分为以下三个阶段：

抽象模型构造阶段是B方法开发过程的第一阶段。它从非形式化描述的软件需求文档中提取需求，并对需求进行精化，加入细节，并加入必要的不变式，最终构造出用B语言描述的系统抽象模型，它是软件需求文档的形式化表示。该阶段分为以下4个步骤：

1. 阅读软件需求文档，总结需求核问题，建立最初始的抽象模型。
2. 向已有抽象模型中不断添加细节，对其进行扩充，并加入必要的不变式。
3. 利用B方法的工具自动抽取需要证明的不变式，这些不变式被称为证明义务。证明义务的证明一些可以由工具自动完成，一些则需要人的交互证明。
4. 重复过程2和3，直到得到系统的一个完整的抽象模型为止，并且所有的不变式都已被证明。

在上述步骤中，初始模型的建立和扩充以及不变式定理的加入都需要人工完成。不变式定理（证明义务）的抽取由B方法工具自动完成。证明义务可以半自动地证明。本阶段的输出是软件系统的一个完整抽象模型。

精化是B方法开发过程的第二阶段。本阶段在第一阶段构造出的系统抽象模型的基础上，向该模型中不断添加细节，对其逐步精化，并对精化的正确性进行证明，最终得到软件的具体模型。该阶段分为以下三个步骤：

1. 向已有模型中添加更多的细节，使更多操作具体化。用更具体、更接近计算机实现的数据结构代替原有模型中抽象的数据结构，得到一个更加具体的模型（依旧是抽象模型）。
2. 证明经过第1步精化的抽象模型是原有抽象模型的精化，即证明精化过程的正确性。经过证明的精化过程才能保证一致性。
3. 不断重复迭代1,2步，精化出一个可以直接对应到实现的模型。这个模型中的数据结构（数据描述）已经是十分具体的数据结构，如数组、树等，而不是集合、映射等抽象数据结构。而且模型中的操作也可以对应到指定的语句结构，如赋值语句、循环语句等。

在上述步骤中，精化工作由人来完成，精化过程正确性的证明义务由B方法工具自动生成，证明义务的证明则可以半自动化完成。

生成可执行代码是B方法开发过程的第三阶段。本阶段检查第二阶段得到的系统具体模型是否已经可以翻译成特定的语言代码，如C代码。如果可以，则使用工具将其翻译到特定的编程语言，并使用编译器生成可执行程序。此阶段可完全自动地完成。

B方法是一种十分重要的形式化方法，已经在很多领域成熟地使用。B方法中没有明显的形式化验证过程，它是通过保证逐步精化的正确性来保证生成代码的正确性。然而，B方法在软件开发过程会产生大量的证明义务，证明义务的证明工作量十分巨大，并且需要多次精化才能得到可执行的系统代码，因此B方法面临着精化过程复杂，证明义务繁重等问题。

### B\*形式化方法

B\*形式化方法是本文所述课题组的研究成果，B\*语言则是B\*方法的规范形式语言。课题组提出B\*方法的动机是为了克服B方法中存在的抽象机精化过程复杂，证明义务繁重等问题，从而更简单高效地用形式化方法来开发和验证软件。B\*将数组、链表、树等数据结构统一抽象成集合，并在集合抽象层次上对软件进行形式验证，在验证正确的基础上，开发代码自动转换工具，将B\*程序转换为可执行代码。本文的工作属于B\*形式化开发方法中的验证工作，即基于B\*语言的形式化验证。

B\*语言主要有三大类数据类型：基本数据类型（如整型、浮点型）、集合和元组数据类型。其中，数组、链表、树等具体数据结构可以使用集合来统一抽象，并且在完备的公理系统和集合论的基础上，B\*语言的表达能力不会降低。

B\*形式化方法采用自顶向下的开发过程，省去了B方法中逐层精化的过程，从而避免了精化过程中产生的大量证明义务。B\*的整个软件开发过程如下：

编写高层抽象机：根据需求文档，定义高层需求，设计软件框架，规划软件的模块构成，规定软件的功能。

编写低层抽象机：根据高层抽象机，构造软件的形式化需求，给出软件每个功能模块（一个功能模块可以理解为一个函数程序）的形态，并给出每个功能模块的形式规约（前置条件和后置条件）。

抽象机实现：根据低层抽象机定义的各功能模块形态，使用B\*语言，以基本数据类型、集合类型和元组类型为基础编写各模块的B\*程序。用户在编写程序时，无需关心具体实现的数据结构，降低了编程的难度。

形式化验证：对B\*语言实现的抽象机使用本文设计和实现的基于语境的形式化验证工具进行验证，推出每个模块程序的功能逻辑表达式，并与对应的抽象机的形式规约进行对比，判定抽象机实现的正确性。

代码转换：在B\*实现的抽象机程序经过形式化验证后，使用B\*代码转换工具，将集合类型使用具体的数据结构，如数组、链表等进行替换，从而生成C代码。

经过验证的B\*程序肯定是满足其形式规约的，而且代码转换使用了替换定理，不会破坏原B\*程序满足形式规约这一性质，因此转换出来的C代码也是满足软件需求的。

本文涵盖了B\*形式化方法中的编写低层抽象机、抽象机实现和形式验证三个过程，主要利用B\*方法中B\*语言作为规范形式语言，编写B\*程序，规定形式规约，为验证工作做准备。因为B\*语言只保留了基本数据结构（如整型、浮点型）、集合和元组数据类型，大大降低了形式验证的难度。同时，B\*语言与C语言类似，而且用户在编写B\*程序时无需考虑具体实现的数据结构，因此容易学习和掌握，降低了验证工具的使用门槛。

## 比较

本章对两种经典的程序正确性证明方法进行了介绍，分析了它们的证明思路并指出面临的问题。Hoare逻辑通过Hoare三元组完成对程序的证明，通过证明每一步中间断言的正确性来证明整个程序的正确性。然而Hoare逻辑证明效率低，过多需要人的参与，且对于循环的证明依赖于循环不变式；归纳构造演算使用lambda表达式对程序进行描述，使用类型系统对程序进行证明。使用归纳构造演算证明程序时，需要证明者具备较高的专业知识，且证明难度较大。当证明循环结构程序时，其也面临着与Hoare逻辑相同的问题。Hoare逻辑与归纳构造演算的比较如表2所示。

表2 Hoare逻辑与归纳构造演算比较

|  |  |  |
| --- | --- | --- |
| **属性** | **Hoare逻辑** | **归纳构造演算** |
| **程序表示** | Hoare三元组 | Lambda表达式 |
| **特点** | 逐步验证程序语句 | 使用递归思想证明程序 |
| **应用** | SCADE，分离逻辑[21] | Coq证明器 |
| **面临问题** | 需要人工构造中间断言  依赖于循环不变式 | 证明过程不直观，理解困难  依赖于循环不变式 |

本章在B方法的基础上，详细介绍了课题组提出的B\*形式化方法，并指明了本文涵盖的B\*方法中的开发阶段以及验证所依赖的形式规范语言B\*语言，对其相对于B方法和B语言的优势进行了分析，为后续章节使用B\*语言作为验证语言奠定了基础。B方法与B\*方法的比较如表3所示。

表3 B方法与B\*方法比较

|  |  |  |
| --- | --- | --- |
| **属性** | **B方法** | **B\*方法** |
| **基础** | 谓词逻辑、集合论 | 谓词逻辑、集合论 |
| **现状** | 已在工业界多个领域成熟使用 | 完善研究阶段 |
| **开发阶段** | 规格说明、设计、实现 | 规格说明、设计、验证、实现 |
| **特点** | 逐步精化  代码自动转换 | 三层抽象机  代码转换使用了替换定理，保证转换一致性 |
| **面临问题** | 形式化描述代价大  逐步精化产生大量证明义务 | 仍需进一步完善 |

# 基于语境的形式化证明方法

本章首先针对于传统形式化验证方法面临的问题和挑战，提出了基于语境的形式化证明思路，将语境的概念和限定数学归纳法引入到本文的形式化证明工作中去。接下来详细介绍了基于语境的形式化证明语义的设计，针对于不同的程序结构，给出了相应的证明语义和转换为一阶逻辑表达式的规则。最后介绍了本文形式化验证所依赖的公理系统以及证明过程中使用的推导规则，即证明策略，并对证明策略的正确性进行了证明。本章是本文的理论基础和依据。

## 基于语境的形式化证明思路

传统的基于定理证明的形式化验证方法一般需要人为构造中间断言，从而降低了程序证明的效率。并且对于循环结构程序的证明，依赖于循环不变式的构造，从而加大了程序证明的难度。本论文针对中间断言构造的问题，提出直接从程序推导出其对应的功能逻辑表达式的思路，引入蒙太古语用学，将语境的概念引入整个软件证明的过程中，即：函数局部变量、全局变量以及上下文构成语境。在不同的语境中，相同变量可以有不同语义。语境记录了程序验证过程中函数变量的状态变化以及变量间的关系，记录了程序证明每一步的程序状态；针对于循环不变式构造困难的问题，将限定数学归纳法引入程序循环结构的逻辑证明，避免了循环不变式的构造，从而降低了循环结构程序证明的难度。

基于语境的形式化证明方法可表述如下：

其中，表示语境，x表示与当前带证明语句相关的变量、常量，k是x所处的状态，代表程序初始语境，是程序，此处要求程序数据结构只含有基本类型和集合类型。Q是程序形式化逻辑功能描述，即程序功能的逻辑表达式。与传统的已知程序功能的逻辑表达式，通过Hoare公理等逐步证明程序中间断言，最后检测其是否满足逻辑功能表达式的方法不同，基于语境的形式化证明方法无需知道功能逻辑表达式Q，而是在已知初始语境以及待验证程序的基础上，逐步从程序进行推导，之后再对推导出的Q的正确性和鲁棒性进行验证。由于在从程序推导出逻辑功能表达式Q的过程中，赋值语句是集合操作，条件选择语句和循环语句变换为逻辑语句的方式确定，因此推导序列基本确定。从而可以实现基于语境的形式化证明方法的自动化（循环结构除外）。此外，同传统的软件验证方法一样，在得到证明序列后可以机械化地验证证明序列的正确性。

Q代表程序的功能逻辑表达式，其形式化描述为：

其中，记为程序的输入空间，记为程序的输出空间，x代表程序的输入，y代表程序的输出，与分别为和的真值函数，分别表示x应满足的关系和x、y应满足的关系。

本文提出使用限定数学归纳法来证明循环结构程序的思路。限定数学归纳法的逻辑基础是自然数公理[22]，也称皮亚诺公理，具体公理如下：

1. 0是自然数，即0N；
2. 若nN，则只存在一个自然数n+称为n的后继数，记为s(n)；
3. ，没有自然数的后继是0；
4. ，对于任意的自然数m，n，如果m的后继等于n的后继，则m等于n；
5. 若子集S N，且具有性质
   1. 0 S；
   2. 如果n S，则s(n) S

则S = N；

公理1~5分别说明：(1) 0是自然数；(2) 每个自然数n都有一个确定的后继s(n)；(3) 没有后继是0的自然数；(4) 每个自然数有且仅有一个后继；(5) 断言满足条件a, b的极小集合是自然数集合，它是数学归纳法的基础。

数学归纳法的表述如下：

设P(n)是自然数集合上的性质（或谓词），如果能够证明：

1. P(0)是真；
2. 对任何mN，P(m) P(s(m));

则对所有nN，P(n)为真。

一般数学归纳法的逻辑表达式为，若P(0)成立且对于任意的自然数n，若P(n)成立则P(s(n))也成立，则对于任意的自然数n，P(n)成立。

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。循环的可终止性由人来证明，终止条件由人给出。

因为去除乘法的自然数公理是完备的，所以得到的功能逻辑表达式

是可判定的。

基于语境的形式化证明思路相对于传统的程序验证方法而言，对于非循环程序可以完全自动地从程序推导出其功能逻辑表达式，无需人为构造中间断言，减少人出错的可能性，并且大幅度提高了证明效率；对于循环结构程序，引入了限定数学归纳法，避免了循环不变式的构造，降低了证明的难度。

## 基于语境的形式化证明语义的设计

基于语境的形式化证明语义规定了将程序逐句转化为基于一阶逻辑和集合论的一阶逻辑表达式的转换规则以及需要满足的约束条件，其作用在于保证了转化前后语义的一致性。

语境记录了程序证明过程中每一步证明项的环境及上下文信息，其包含以下内容：

1. 全局域（G）：当前证明项相关的全局常量以及全局变量；
2. 局部域（L）：当前证明项相关的局部常量以及局部变量；
3. 参数（P）：当前证明项相关的形参；

不同于符号表，语境不仅记录了变量的真实值（如果变量的值可计算），而且记录了变量的状态变化以及变量间的关系，这有助于用户更好地理解程序状态和证明过程，更容易地发现循环中变量的递推关系，从而降低交互证明的难度。

令w为待证明命题的语境，为待证明的语句序列，则形式化证明的目的如下：

)

对于不同的程序结构，我们给出相应的证明语义和转换规则，具体规则如下：

1. 并置规则

并置规则为我们顺序证明程序语句提供了理论基础。

1. 赋值规则

表示将exp中的x使用x的当前语境替换。

1. 条件选择规则

条件选择语句需要证明条件Q成立以及Q不成立的命题均为真命题。

1. 循环规则

循环规则可以理解为将循环结构的程序拆分成为一个条件结构（假设条件Q不会改变参数x的语境）和执行完一次循环体后的循环结构。该规则不依赖于循环不变式。

1. 函数调用规则

Q1，Q2等代表被调用函数func中的语句。函数调用规则相当于将被调用函数展开，并利用并置规则逐句证明函数体中的语句序列。

1. 返回规则

返回语句代表程序（本文一个程序指一个函数）验证或者某个分支验证的结束。函数的作用在于改变的全局变量的值、参数的值以及返回值，因此在程序验证结束时，改变了的全局变量和函数参数也应作为函数功能逻辑表达式中的一部分。

在上述规则中，横线上方是给定的语境和程序结构，横线下方是该程序结构转化到的一阶逻辑表达式。

在给定语义转换规则后，便将程序正确性的证明转换为了逻辑推导，并可以使用推导规则自动推导，无需人为构造中间断言，从而提高了证明的效率；对于循环语义转换规则，其逻辑表达式中不包含循环不变式，避免了循环不变式的构造，从而降低了证明的难度。

## 证明策略

程序的形式化证明需要特定的公理系统作为基础。所谓公理系统[22,23]，是指从事先给定的公理出发，根据推理规则推导出一系列定理，由此而形成的演绎系统。公理系统一般由符号集、公式集（用于表达命题的符号串）、公理集（永真式集的真子集，公理是用于表达推理由之出发的初始肯定命题）、推理规则集（推理规则是由公理及已证定理得出新定理的规则）和定理集（表达本系统肯定的所有命题）组成。

证明策略的本质是形式化证明所依赖的公理系统中的推理规则集，即由公理及已证明的定理得出新定理的规则。推理规则集满足公理系统的可靠性和完备性，为程序证明中使用证明策略提供了理论基础。

本文的程序形式化证明依赖的是基于一阶逻辑的公理系统，同时融入集合论中不破坏可靠性和完备性的公理。证明策略则由Read规则、MP规则、Split规则和Combine规则组成。我们对这四个规则的正确性分别进行说明，设A、B、C为命题：

Read规则根据形式化证明语义中的转换规则将不同结构的程序语句转化为相应的一阶逻辑表达式。

MP规则是一阶逻辑公理系统自带的推理规则，又称分离规则。一阶逻辑的极小完备系统可以由公理与MP规则给出。MP规则可表示为。具体证明如下：

1. —— 前提
2. —— 演绎定理

Split规则主要用于条件选择语句、循环语句和函数（包含条件选择语句和循环语句）调用语句的证明中，即将语义转换后得到的合取式的证明转化为各条件分支的证明。Split规则可表示为：。具体证明如下：

1. —— 前提
2. —— 前提
3. —— 合并

Combine规则将多重蕴含关系转化为多个前提的合取式与结论的蕴含关系，为每一个证明项提供完整前提，主要用于多重条件选择语句、循环语句中。Combine规则可表示为：。具体证明如下：

1. 当且仅当
2. 当且仅当 —— 演绎定理
3. 当且仅当 —— Split规则
4. 当且仅当 —— MP规则
5. —— MP规则

证毕 —— 结论包含于前提

在本文提出的程序形式化证明过程中，会自动选择证明策略并根据相应策略将某一命题拆分、合并或者转化为新命题，并作为前提应用到之后的程序证明过程中。因为只能使用已经证明正确的证明策略进行推导，所以保证了证明序列中的每一项都是前提、公理或者定理，又因为公理系统是可靠的，所以证明序列中的每一项一定是正确的，从而最终推导出来的程序功能逻辑表达式一定是正确的。

## 小结

本章针对于传统形式化证明方法面临的问题，提出了基于语境的形式化证明方法，提高了证明效率，降低了证明难度，具体如下：

1. 针对于传统形式化证明方法中因中间断言构造导致证明效率低的问题，本章提出了一种新的形式化证明思路，即以推导的方法代替逐步验证的方法，并基于此思路，引入了语境的概念，设计了基于语境的形式化证明语义，给出了各程序结构转换为一阶逻辑表达式的转换规则，使得程序证明转换为逻辑推导，无需人工构造中间断言，从而提高了证明效率。
2. 针对于传统形式化证明方法中因依赖于循环不变式导致证明难度大的问题，本章提出的基于语境的形式化证明方法跳出了遇循环结构程序便寻找循环不变式的思想禁锢，使用限定数学归纳法来证明循环结构程序，避免了循环不变式的构造，从而降低了证明难度。
3. 基于语境的形式化证明方法以一阶逻辑公理系统为基础，基于语境的语义转换规则作为非逻辑公理融入到一阶逻辑的公理系统中。程序证明过程只使用已证明的推导规则进行逻辑推导，保证了证明过程的正确性，从而保证了推导出的功能逻辑表达式的正确性。

# 基于语境的形式化证明算法

本章设计了基于语境的形式化证明算法，重点阐述了基于语境和限定数学归纳法的循环交互证明算法。本章提出的证明算法是对第三章中语境概念、基于语境的形式化证明语义规则和证明策略的算法设计。

## 语境算法

语境记录了程序证明过程中每一步证明项的环境及上下文信息，记录了各变量、常量以及参数的真实值（如果可以计算）以及变量间的关系，是推动程序证明的重要基础。

语境算法主要是语境提供、语境计算和语境更新。语境算法仅提供与当前证明项相关的语境，不提供访问不到的语境，用于语境计算；语境更新是当证明遇到赋值语句时，就应该在计算右表达式值后更新被赋值变量的语境；语境算法的核心部分是语境计算，即表达式求值，主要分为以下两种情况：

1. 如果表达式中存在未被赋值的变量或者常量，则仍以其标识符表示，表达式值表现为标识符运算组合。
2. 如果表达式中所有变量已知其真值，则计算相应表达式的真值，主要运算符包括算数运算如+、-、\*、/以及集合运算等等。

给定赋值语句的语法树结点，抽取等号右边的表达式结点，对该语法树结点进行深度遍历，得到的遍历序列即为表达式的前缀表达式（波兰式）。对前缀表达式的计算如下：从后向前扫描前缀表达式，并设定一个操作数栈，如果是操作数(在本文中可以是真实值或者表达式运算组合)，则将其直接入栈，如果是操作符，则从栈中弹出操作符对应的操作数进行运算，并将计算结果压栈。直至从右到左扫描完毕整个前缀表达式，这时操作数栈中应该只有一个元素，该元素的值则为前缀表达式的计算结果。

语境计算算法如表4所示，其输入为表达式的语法树结点，输出为表达式的值。

表4 语境计算算法

|  |
| --- |
| **FunctionName: Caculate\_Expression**  **Input:** Expression\_Node  **Output:** Expression\_Value |
| 1: **Def**: **Caculate\_Expression** (Expression\_Node)  // 深度遍历要计算值的表达式结点，生成前缀表达式  2: Pre\_Expression = **DFS**(Expression\_Node)  // 逆序遍历前缀表达式，并按照规则出栈和压栈 |

表5 语境计算算法（续）

|  |
| --- |
| 3: **For** element **in** **Reverse**(Pre\_Expression):  // 如果是操作数，则将操作数压栈  4: **case:** element **is** expression:  5: Stack.push(element)  // 如果是运算符，则根据运算符对应的操作数数目出栈元素，并将  运算结果压栈  6: **case:** element **is** operator:  7: **for** i **from** 1 to **Get\_Oper\_Num**(element):  8: List.push\_back(Stack.pop())  9: result = **caculate**(List, element)  10: Stack.push(result)  11: return Stack.pop()  12: **End** |

## 基于语境的语义规则映射算法的设计

对给定的待验证程序进行词法分析和语法分析，便可得到源程序相应的语法树。映射算法的输入即为待验证程序的语法树。映射算法根据语句节点类型分为赋值规则映射算法、条件选择语句规则映射算法、循环语句映射算法、返回语句映射算法和函数调用语句映射算法。

程序验证的过程即是按照映射规则遍历语法树的过程，程序验证过程中每次从程序语法树中读取一条语句结点，然后判断该语句的语句类型，并根据相应的语义转换规则（Read规则）生成初始证明项，再利用证明策略（MP规则、Split规则和Combine规则）对证明项命题进行拆分、合并等操作，推动程序证明，直到最后一个语句结点被验证完毕。

首先给出整个程序的证明算法Proof\_Program，随后给出不同语句结构的相应映射算法。算法Proof\_Program如表6所示。算法的输入是整个程序语法树，算法的输出是程序的证明序列以及程序的功能逻辑表达式。算法在第9行和第16行使用了Break语句去终止第2行的循环，因为当遇到条件选择语句结点和函数调用语句结点时，对应的规则映射算法会有拼接剩余语句结点并进行验证的操作，因此无需再验证当前语句结点之后的语句，即应当使用Break语句跳出循环。

表6 程序证明算法Proof\_Program

|  |
| --- |
| **FunctionName: Proof\_Program**  **Input:** Syntax\_Tree  **Output:** Function\_Expression, Proof\_Sequences //程序功能表达式、证明序列 |
| 1: **Def:** **Proof\_Program** (Syntax\_Tree) |

表7 程序证明算法Proof\_Program（续）

|  |
| --- |
| 2: **For** **Every** Statement **in** Syntax\_Tree：  3: // 判断每个结点的类型  4:  **Switch**(Statement)：  5: **case:** Assignment Statement // 赋值语句  6: **Proof\_Assignment**(Syntax\_Tree, Statement)  7: **case:** Selection Statement // 条件选择语句  8: **Proof\_Selection**(Syntax\_Tree, Statement)  9: **Break**  10: **case:** While\_Statement // 循环语句  11: **Proof\_While**(Syntax\_Tree, Statement)  12: **case:** Return Statement // 返回语句  13: **Proof\_Return**(Syntax\_Tree, Statement)  14: **case:** Function\_Call Statement // 函数调用语句  15: **Proof\_Function\_Call**(Syntax\_Tree, Statement)  16: **Break**  17: **End** |

### 赋值规则映射算法

赋值规则映射算法对应于语义转换规则中的赋值规则，用于证明赋值语句，对应于算法Proof\_Assignment，如表8所示。首先使用语境计算算法计算出赋值语句等号右边表达式的值，然后提取出等号左边的变量，最后将表达式的值赋予变量，即更新变量的语境，并将语境的改变记录到语境表中。赋值语句只进行语境计算和语境更新，不生成证明项。赋值规则映射算法的输入是程序语法树和赋值语句结点，输出为空，算法会更新语境。

表8 赋值语句映射算法Proof\_Assignment

|  |
| --- |
| **FunctionName: Proof\_Assignment**  **Input:** Syntax\_Tree, Assignment\_Statement  **Output:** null |
| 1: **Def**: **Proof\_Assignment** (Syntax\_Tree, Assignment\_Statement)  // 得到赋值语句中等号右边的表达式  2: Right\_Expression = **Get\_Right**(Assignment\_Statement)  // 计算Right\_Expression表达式的值  3: Right\_Value = **Caculate\_Expression\_Value**(Right\_Expression)  // 得到赋值语句中等号左边变量  4: Left\_Variable = **Get\_Left**(Assignment\_Statement)  // 根据Left\_Variable的类型（表达式、数值、指针）来更新相应语境  // 如果是指针，还需要相应更新其所指向的变量  // 如果是元组中某一维，则也会引起元组整体语境的变化  5: **Update\_Context**(Left\_Variable, Right\_Value)  6: **End** |

### 条件选择规则映射算法

条件选择规则映射算法对应于语义转换规则中的条件选择规则，用于证明条件选择语句，对应于算法Proof\_Selection，如表9所示。对于形如的条件选择语句，根据公理系统中的条件选择规则，将条件选择语句转换为形式为的逻辑表达式，再自动应用证明策略中的Split规则，将证明分成Q成立和Q不成立两个分支去证明。算法需对Q中的变量或常量代入当前语境，计算Q的真伪（如果Q中变量真值都已知），并根据Q的真伪去选择验证分支。如果Q真值不可计算，则将Q的逻辑表达式作为前提，验证Q成立和Q不成立两个分支。条件选择映射算法的输入是程序语法树和条件选择语句结点，输出为条件选择结点的证明项。

表9 条件选择语句映射算法Proof\_Selection

|  |
| --- |
| **FunctionName: Proof\_Selection**  **Input:** Syntax\_Tree, Selection\_Statement  **Output:** Selection\_Proof\_Item |
| 1: **Def**: **Proof\_Selection** (Syntax\_Tree, Selection\_Statement)  // 根据节点类型和对应的语义转换规则生成证明步  2: Proof\_Item = **Generate\_Proof\_Item** (Selection\_Statement))  3: **Print**(Proof\_Item)  // 根据Split规则，将程序证明分为两个分支  4: Right\_Expression = **Get\_Right**(Assignment\_Statement)  // 得到该语句后程序的所有剩余语句，记为Remain  5: Remain = **Get\_Remain\_Statements**(Syntax\_Tree,Selection\_Statement)  // 得到If体中的语句  6: Ifs\_Statements = **Get\_Ifs**(Syntax\_Tree, Selection\_Statement)  // 用Ifs\_Statements和Remain构造if条件成立的分支语法树  7: Ifs\_Syntax\_Tree = **Construct\_New\_Tree**(Ifs\_Statements, Remain)  // 备份语境表  8: Context\_Table\_Record = Content\_Table  // 递归证明if条件成立分支  9: **Proof\_Program**(Ifs\_Syntax\_Tree)  // 恢复语境表，证明Else分支  10: Content\_Table = Context\_Table\_Record  // 得到Else体中的语句  11: Elses\_Statements = **Get\_Elses**(Syntax\_Tree, Selection\_Statement)  // 用Elses\_Statements和Remain构造if条件不成立的分支语法树  12: Elses\_Syntax\_Tree = **Construct\_New\_Tree**(Elses\_Statements, Remain)  // 递归证明if条件不成立分支  13: **Proof\_Program**(Elses\_Syntax\_Tree)  14: **End** |

### 循环规则映射算法

循环规则映射算法对应于语义转换规则中的循环规则，用于证明循环语句，对应于算法Proof\_While，即基于限定数学归纳法的循环交互证明算法。该算法在本章4.3节中单独给出。

### 函数调用规则映射算法

函数调用规则映射算法对应于语义转换规则中的函数调用规则，用于证明函数调用语句，对应于算法Proof\_Function\_Call。被调用函数所在的抽象机会在当前抽象机的Include部分进行声明，用于在发生函数调用时可以追溯到其所在抽象机并获取程序。对应算法如表10所示。

表10 函数调用规则映射算法Proof\_Function\_Call

|  |
| --- |
| **FunctionName: Proof\_Function\_Call**  **Input:** Syntax\_Tree, Function\_Call\_Statement  **Output:** Function\_Call\_Proof\_Item |
| 1: **Def**: **Proof\_Function\_Call** (Syntax\_Tree, Function\_Call\_Statement)  // 得到被调函数的函数名  2: Function\_Name = **Get\_Function\_Name**(Function\_Call\_Statement)  // 得到被调函数的程序  3: Function\_Program = **Get\_Function\_Program**(Function\_Name)  // 生成被调函数的语法树  4: Function\_Syntax\_Tree = **Generate\_Syntax\_Tree**(Function\_Program)  // 获取函数调用语句结点后的剩余语句结点  5: Remain = **Get\_Remain\_Statements**(Syntax\_Tree, Function\_Call\_Statement)  // 拼接被调函数语法树以及剩余语法树  6: Function\_Remain\_Combine = **Construct\_New\_Tree**(Function\_Syntax\_Tree, Remain)  //使用程序证明算法递归证明  5: **Proof\_Program** (Function\_Remain\_Combine) |
| 7: **End** |

### 返回规则映射算法

返回规则映射算法对应于语义转换规则中的返回规则，由于返回语句是最后一条语句，此步得到的证明项便是程序的功能逻辑表达式。如果程序包含多重条件选择语句或者循环语句，则需要利用Combine规则将所有前提合取，共同推出函数的功能逻辑表达式。并且，程序的功能逻辑表达式应包括全局变量语境的改变和参数语境的改变。映射方法对应于算法Proof\_Return，如表11所示。算法的输出是程序的证明序列和功能逻辑表达式。

表11 返回语句映射算法Proof\_Return

|  |
| --- |
| **FunctionName: Proof\_Return**  **Input:** Syntax\_Tree, Return\_Statement  **Output:** Function\_Exp, Proof\_Sequences //程序功能表达式、证明序列 |
| 1: **Def**: **Proof\_Return** (Syntax\_Tree, Return\_Statement)  // 获取所有语境改变的全局变量和参数，作为部分函数功能逻辑表达式  2: Part\_Function\_Exp = **Get\_All\_Changed\_Context**(Context\_Table)  // 生成证明项  3: Function\_Exp = **Generate\_Proof\_Item** (Return\_Statement)  // 此时得到的证明项是程序功能逻辑表达式  4: **Record\_Function\_Expression**(Proof\_Step)  5:  **Print**(Function\_Exp) // 输出程序功能的逻辑表达式  6: **Print**(Proof\_Sequences) // 输出证明序列  7: **End** |

## 基于语境和限定数学归纳法的循环交互证明算法的设计

针对于传统程序验证方法中循环不变式构造没有一般方法，且构造困难的问题，本文提出基于限定数学归纳法的循环交互证明算法，即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，引导用户使用数学归纳法对循环进行归纳总结，并借助语境，帮助用户了解变量间关系和程序证明状态，从而给出循环的功能逻辑表达式和终止条件。对应算法如表12所示，该算法的证明步骤如下：

1. 根据公理系统规则映射算法，逐次执行循环内的语句块，推导出循环结构前有限步数的证明项，引导用户给出循环结构在第N次执行后的功能逻辑表达式。
2. 提示用户输入循环结构在第N次执行后的功能逻辑表达式User\_Expression。
3. 以User\_Expression为基础，临时更新语境表，并在最新语境的基础上再执行一次循环体（根据公理系统规则映射算法），执行结束后，得出验证工具自动推导出的该循环语句的在第N+1次执行后的功能逻辑表达式Inference\_Expression。
4. 将User\_Expression中的关于N的项使用N+1进行替换，即将User\_Expression扩展为关于N+1的循环的功能逻辑表达式。
5. 验证Inference\_Expression与User\_Expression的等价性，如果二者等价，则用户输入的循环结构的功能逻辑表达式是正确的；否则回滚循环证明，提示用户重新证明。

表12 循环语句映射算法Proof\_While

|  |
| --- |
| **FunctionName: Proof\_While**  **Input:** Syntax\_Tree, While\_Statement |

表13 循环语句映射算法Proof\_While（续）

|  |
| --- |
| **Output:** While\_Function Expression // 输出是循环语句的功能逻辑表达式 |
| 1: **Def**: **Proof\_While** (Syntax\_Tree, While\_Statement)  // 生成该循环结构前有限次数的证明项  2: **For** i **from** 1 **to** Specific\_Steps:  3: **Generate\_Proof\_Item** (While\_Statement)  // 读取用户输入的关于循环第N次执行后的功能逻辑表达式的假设  4: User\_Expression = **Read\_User\_Input\_Expression**()  // 以用户输入为基础，临时更新语境  5: **Temp\_Update\_Context**(User\_Expression)  // 以User\_Expression为和最新语境为基础，再次执行循环体语句  6:  **For** **Every** Statement **in** While\_Statement：  7: Inference\_Expression = **Generate\_Proof\_Item**(Statement)  // 将User\_Expression中的关于N的项使用N+1进行替换  8: **Expand\_User\_Input\_Expression**(User\_Expression)  // 验证Inference\_Expression与User\_Expression的等价性  9: **If**(**Expression\_Is\_Equivalent**(Inference\_Expression,User\_Expression)  10: **return** User\_Function\_Expression  11: **Else**:  // 回滚循环证明，即恢复循环证明开始时的语境状态  12: **Roll\_Back**()  // 重新证明循环语句结点  13: **Proof\_While** (Syntax\_Tree, While\_Statement)  14: **End** |

## 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。

# 基于语境的形式化验证工具的设计与实现

本章介绍了基于语境的形式化验证工具的从需求分析到代码实现的整个工具开发过程，是对第四章的基于语境的形式化证明算法的分析与实现。本章最后给出一个循环结构程序验证的应用示例用于说明验证工具的证明形态。

## 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。

# 基于B\*的Arinc653内核建模与验证

本章首先介绍了Arinc653标准，然后详细阐述了使用B\*语言对Arinc653内核函数进行建模以及使用基于语境的形式化验证工具对内核函数抽象机进行验证的过程，最后对验证结果进行了分析，并与现有的L4内核验证案例进行比较。

## Arinc653标准

ARINC653标准[24,25](Avionics Application Software Standard Interface)是由ARINC公司于1997年发布的航空电子应用软件标准接口规范。该规范定义了标准的API和系统服务，即应用/执行（APplicationEXecutive, APEX）层，把应用程序和操作系统隔离，还要求通过时间分区、空间分区以保证将不同的应用的隔离。每个分区的底层架构类似于一般意义上的大型计算机的多任务应用的机制。每个分区有一个或多个并行的进程，按照应用的资源使用要求对该进行所拥有的资源实行共享访问。ARINC653是工业界普遍认可的标准，

ARINC653体系结构主要分为应用软件层和核心软件层两层，其中应用软件层又可细分为应用分区和系统分区两部分。应用分区主要是指航电应用系统的软件部分，这些系统都要根据航电应用的安全级别进行认证。系统分区指APEX接口所提供的功能范围之外的服务，并且受ARINC653的时间和空间分区限制的分区。系统分区主要负责硬件设备的通信和容错机制。系统分区是可选的，并且可根据具体的核心模块实现指定。核心软件层主要包括操作系统内核和系统指定功能两部分。操作系统内核主要提供ARINC653规范的功能以及系统运行的统一的共享环境。系统指定功能主要提供诸如设备驱动、上传下载、调试等功能的硬件接口。

应用/执行（APEX）接口，位于ARINC653规范中的应用软件层和核心软件层之间，其目的是为基于IMA的应用软件和操作系统之间提供一个通用目的的接口，保障在同一个硬件平台上的各个航电系统或系统的不同功能模块可以独立运行。

ARINC653规范主要提供的服务有：分区管理、进程管理、时间管理、内存分配、分区间通信、分区内通信、健康监控等。本课题组对分区管理、进程管理、时间管理、分区间通信、分区内通信和健康监控等规范使用B\*语言描写，并进行了验证。ARINC653主要规范模块如下：

分区管理：分区是ARINC653规范的核心概念，依照IMA规范，分区就是一个拥有独立的内存空间和严格的保护时间的运行环境，无论在任何情况下，都不会影响其他分区。分区根据配置文件中的调度计划进行循环调度。另外，分区所使用的资源都必须在系统配置时进行定义，并且在分区初始化期间进行创建和初始化。

进程管理：一个分区内包含一个或多个进行动态交互的进程来完成分区的功能。进程是ARINC653规范中的分区的可执行单元。进程调度器按照优先级抢占的策略进行调度。通过这种方式对分区内的进程进行管理。进程是分区内唯一可见单元。

分区间通信：该模块主要定义了两个或者多个分区间的消息传递的通信机制。分区间通信的基本机制就是通道。通道定义了一个信息源与一个或多个目的端之间的逻辑链接。端口允许指定的分区从通道中读取或写入信息。定义通道通信时需要定义通道、端口、最大信息长度、最大信息个数等信息。分区间通信主要有队列通信和采样通信两种方式。队列端口通信方式允许源端口发送消息队列给目的端口，而采样通信则是源端口和目的端口共享消息变量。

分区内通信：分区内通信定义了同一个分区内的进程间的消息通信和同步方式。分区内的进程之间的信息同步主要有信号量和事件同步两种方式。分区内的通信方式有缓冲区（共享消息队列）和黑板（共享消息变量）两种方式。缓冲区以消息队列的方式存储多个消息，每个消息都带有唯一的数据，传送时不允许覆盖。对于黑板来说，消息队列是不允许的，任意写到黑板的消息将一直保留到被清除或者被新的消息覆盖。

## Arinc653内核建模

### Arinc653高层抽象机建模

高层抽象机建模是指根据需求文档，定义高层需求，设计软件框架，规划软件的模块构成，规定软件的功能。Arinc653规范定义了分区管理、进程管理、时间管理、内存管理、分区间通信、分区内通信和健康监控服务需求。其中，分区间通信需求包括采样端口服务需求和队列端口服务需求。分区内通信服务包括缓冲区服务需求、黑板服务需求、信号量服务需求和事件服务需求。

本课题组根据Arinc653规范，设计了与各需求对应（内存管理模块除外）的抽象机模块，即PartitionManagement（分区管理）、ProcessManagement（进程管理）、SamplingManagement（采样端口服务）、QueueManagement（队列端口服务模块）、TimeManagement（时间管理）、BufferManagement（缓冲区服务模块）、BlackboardManagement（黑板服务模块）、SemaphoreManagement（信号量服务模块）、EventManagement（事件管理）、HealthMonitor（健康监控），并对各抽象机的功能和使用的数据类型进行规定。

下面以进程管理模块为例来阐述使用B\*对ARINC653内核函数进行建模及验证的过程。根据Arinc653需求文档，定义了进程管理模块用到的数据结构，具体如表28所示，并在定义的数据结构的基础上，对Arinc653文档中关于进程管理模块的功能需求使用相应函数进行定义，并与数据结构一起构成了进程管理模块的高层抽象机，具体如表29所示。

表28 进程管理模块数据结构定义

|  |  |  |  |
| --- | --- | --- | --- |
|  | **数据结构** | **数据描述** | **数据类型** |
|  | *PROCESS\_ID\_TYPE* | 进程ID类型 | 数字 |
|  | *PROCESS\_NAME\_TYPE* | 进程名类型 | 字符串 |
|  | *PRIORITY\_TYPE* | 进程优先级类型 | 数字 |
|  | *STACK\_SIZE\_TYPE* | 栈大小类型 | 数字 |
|  | *LOCK\_LEVEL\_TYPE* | 锁级别类型 | 数字 |
|  | *PROCESS\_STATE\_TYPE* | 进程状态类型 | 枚举 |
|  | *DEADLINE\_TYPE* | 截止时间类型 | 枚举 |
|  | *PROCESS\_ATTRIBUTE\_TYPE* | 进程属性类型 | 元组 |
|  | *PROCESS\_STATUS\_TYPE* | 进程状态类型 | 元组 |
|  | *RETURN\_CODE\_TYPE* | 返回状态码类型 | 枚举 |
|  | *SYSTEM\_TIME\_TYPE* | 系统时间类型 | 枚举 |

表29 进程管理模块B\*高层抽象机

|  |
| --- |
| **MACHINE** ProcessManagement  **ATTRIBUTES**  **typedef** int PROCESS\_ID\_TYPE;  **typedef** string PROCESS\_NAME\_TYPE;  **typedef** int PRIORITY\_TYPE;  **typedef** int STACK\_SIZE\_TYPE;  **typedef** int LOCK\_LEVEL\_TYPE;  **typedef** enum  { DORMANT, READY, RUNNING, WAITING}  PROCESS\_STATE\_TYPE;  **typedef** enum  { SOFT, HARD}  DEADLINE\_TYPE;  **typedef** struct {  PROCESS\_NAME\_TYPE NAME;  STACK\_SIZE\_TYPE STACK\_SIZE;  PRIORITY\_TYPE BASE\_PRIORITY;  SYSTEM\_TIME\_TYPE PERIOD;  SYSTEM\_TIME\_TYPE TIME\_CAPACITY;  DEADLINE\_TYPE DEADLINE; |

表30 进程管理模块B\*高层抽象机（续1）

|  |
| --- |
| } PROCESS\_ATTRIBUTE\_TYPE;  **typedef** struct {  PROCESS\_ATTRIBUTE\_TYPE ATTRIBUTES;  PRIORITY\_TYPE CURRENT\_PRIORITY;  SYSTEM\_TIME\_TYPE DEADLINE\_TIME;  } PROCESS\_STATUS\_TYPE;    **typedef** enum  { INVALID\_PARAM, INVALID\_CONFIG, NO\_ACTION, NO\_ERROR,  NOT\_AVAILABLE, TIMED\_OUT, INVALID\_MODE }  RETURN\_CODE\_TYPE;  typedef long long SYSTEM\_TIME\_TYPE;    **OPERATIONS**  RETURN\_CODE\_TYPE **GET\_PROCESS\_ID** (  **/\*in \*/** PROCESS\_NAME\_TYPE PROCESS\_NAME,  **/\*out\*/** PROCESS\_ID\_TYPE PROCESS\_ID );  RETURN\_CODE\_TYPE **GET\_PROCESS\_STATUS** (  **/\*in \*/** PROCESS\_ID\_TYPE PROCESS\_ID,  **/\*out\*/** PROCESS\_STATUS\_TYPE PROCESS\_STATUS);  RETURN\_CODE\_TYPE **CREATE\_PROCESS** (  **/\*in \*/** PROCESS\_ATTRIBUTE\_TYPE ATTRIBUTES,  **/\*out\*/** PROCESS\_ID\_TYPE PROCESS\_ID);  RETURN\_CODE\_TYPE **SET\_PRIORITY** (  **/\*in \*/** PROCESS\_ID\_TYPE PROCESS\_ID,  **/\*in \*/** PRIORITY\_TYPE PRIORITY);  RETURN\_CODE\_TYPE **SUSPEND\_SELF** (  **/\*in \*/**  SYSTEM\_TIME\_TYPE TIME\_OUT);  RETURN\_CODE\_TYPE **SUSPEND** (  **/\*in \*/**  PROCESS\_ID\_TYPE PROCESS\_ID);  RETURN\_CODE\_TYPE **RESUME** (  **/\*in \*/**  PROCESS\_ID\_TYPE PROCESS\_ID);  void **STOP\_SELF**();  RETURN\_CODE\_TYPE **STOP** (  **/\*in \*/** PROCESS\_ID\_TYPE PROCESS\_ID); |

表31 进程管理模块B\*高层抽象机（续2）

|  |
| --- |
| RETURN\_CODE\_TYPE **START** (  **/\*in \*/**  PROCESS\_ID\_TYPE PROCESS\_ID);  RETURN\_CODE\_TYPE **DELAYED\_START** (  **/\*in \*/**  PROCESS\_ID\_TYPE PROCESS\_ID,  **/\*in \*/** SYSTEM\_TIME\_TYPE DELAY\_TIME);  RETURN\_CODE\_TYPE **LOCK\_PREEMPTION** (  **/\*out\*/** LOCK\_LEVEL\_TYPE LOCK\_LEVEL);  RETURN\_CODE\_TYPE **UNLOCK\_PREEMPTION** (  **/\*out\*/**  LOCK\_LEVEL\_TYPE LOCK\_LEVEL);  RETURN\_CODE\_TYPE **GET\_MY\_ID** (  **/\*out\*/**  PROCESS\_ID\_TYPE PROCESS\_ID);  **END** |

进程管理模块的B\*高层抽象机定义了数据结构，规定了数据结构与具体数据类型的对应关系，并将文档中的功能需求使用函数进行形式化描述，并对函数的输入参数、输出参数以及返回值进行了定义。在高层抽象机的基础上，对其各个函数的形式规约进行规定，便可完成低层抽象机的构建。

### Arinc653低层抽象机建模

低层抽象机建模根据需求文档和高层抽象机，给出每个模块中各函数的形式规约，即前置条件和后置条件，并补充需要使用到的其他抽象机模块的数据结构。下面以进程管理模块中的SUSPEND函数为例，详细说明低层抽象机的建模过程。SUSPEND函数的具体需求说明如表32所示。

表32 SUSPEND函数需求说明

|  |  |
| --- | --- |
| **函数名** | SUSPEND |
| **功能描述** | 根据进程号挂起进程 |
| **函数原型** | RETURN\_CODE\_TYPE CREATE\_SAMPLING\_PORT (  /\*in \*/ PROCESS\_ID\_TYPE PROCESS\_ID) |
| **输入参数** | PROCESS\_ID /\* 进程号 \*/ |
| **输出参数** | 空 |
| **返回值** | RETURN\_CODE /\* 返回码 \*/ |
| **正常处理** | 如果指定进程已经处于挂起状态，则函数返回NO\_ACTION; |

表33 SUSPEND函数需求说明（续）

|  |  |
| --- | --- |
|  | 否则将该进程添加至等待队列，函数返回NO\_ERROR; |
| **错误处理** | 如果抢占不允许并且指定的进程号是错误管理处理器已经抢占的进程，则函数返回INVALID\_MODE;  如果指定进程不存在或者指定进程是当前进程，则函数返回INVALID\_PARAM;  如果指定进程的进程状态是DORMANT（休眠），则函数返回INVALID\_MODE;  如果指定进程是周期性进程，则函数返回INVALID\_MODE; |

根据SUSPEND函数需求说明中的正确处理和错误处理，补充了额外的数据结构 PARTITION\_STATUS\_TYPE（分区状态类型），定义了新的PROCESS\_TYPE数据结构，并对错误处理写出了鲁棒性需求的前置条件和后置条件，对正确性处理写出了函数功能性需求的前置条件和后置条件，SUSPEND函数对应的B\*低层抽象机具体如表34所示。

表34 SUSPEND函数B\*低层抽象机

|  |
| --- |
| **MACHINE** ProcessManagement  **ATTRIBUTES**  **#define** INFINITE\_TIME\_VALUE -1;  **typedef** struct {  PROCESS\_ID\_TYPE ID;  PROCESS\_STATUS\_TYPE PROCESS\_STATUS;  PROCESS\_STATE\_TYPE PROCESS\_STATE;  } PROCESS\_TYPE;  PROCESS\_TYPE CURRENT\_PROCESS;  PROCESS\_ID\_TYPE ERROR\_HANDLER\_PROCESS\_ID;  set< PROCESS\_TYPE > PROCESS\_SET;  set<PROCESS\_TYPE> READY\_PROCESS\_SET;  set<PROCESS\_TYPE> WAITING\_PROCESS\_SET;  PARTITION\_STATUS\_TYPE CURRENT\_PARTITION\_STATUS;    **OPERATIONS**  RETURN\_CODE\_TYPE **SUSPEND** (  /\*in \*/ PROCESS\_ID\_TYPE PROCESS\_ID)  **ROBUST\_DEMAND //鲁棒性需求**  **Precondition\_1**  CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID  **Postcondition\_1** |

表35 SUSPEND函数B\*低层抽象机（续1）

|  |
| --- |
| CREATE\_SAMPLING\_PORT == INVALID\_MODE    **Precondition\_2**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) && Pro.<ID:PROCESS\_ID>:PROCESS\_SET &&  Pro.ID == CURRENT\_PROCESS.ID  **Postcondition\_2**  CREATE\_SAMPLING\_PORT == INVALID\_PARAM    **Precondition\_3**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) && Pro.<ID:PROCESS\_ID>:PROCESS\_SET &&  ~(Pro.ID == CURRENT\_PROCESS.ID) &&  Pro.PROCESS\_STATE == DORMANT  **Postcondition\_3**  CREATE\_SAMPLING\_PORT == INVALID\_MODE    **Precondition\_4**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) && Pro.<ID:PROCESS\_ID>:PROCESS\_SET &&  ~(Pro.ID == CURRENT\_PROCESS.ID) &&  ~(Pro.PROCESS\_STATE == DORMANT) &&  Pro.PERIOD != INFINITE\_TIME\_VALUE  **Postcondition\_4**  CREATE\_SAMPLING\_PORT == INVALID\_MODE    **Precondition\_5**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) &&  ~(Pro.<ID:PROCESS\_ID>:PROCESS\_SET)  **Postcondition\_5**  CREATE\_SAMPLING\_PORT == INVALID\_PARAM  **FUNCTION\_DEMAND //功能性需求**  **Precondition\_6**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) &&  Pro.<ID:PROCESS\_ID>:PROCESS\_SET &&  ~(Pro.ID ==CURRENT\_PROCESS.ID)  && ~(Pro.PROCESS\_STATE == DORMANT) && |

表 36 SUSPEND函数B\*低层抽象机（续2）

|  |
| --- |
| ~(Pro.PERIOD != INFINITE\_TIME\_VALUE) &&  Pro.PROCESS\_STATE == WAITING  **Postcondition\_6**  CREATE\_SAMPLING\_PORT == NO\_ACTION    **Precondition\_7**  ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 &&  PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID) &&  Pro.<ID:PROCESS\_ID>:PROCESS\_SET &&  ~(Pro.ID == CURRENT\_PROCESS.ID) &&  ~(Pro.PROCESS\_STATE == DORMANT) &&  ~(Pro.PERIOD != INFINITE\_TIME\_VALUE) &&  ~(Pro.PROCESS\_STATE == WAITING)  **Postcondition\_7**  CREATE\_SAMPLING\_PORT == NO\_ERROR &&  READY\_PROCESS\_SET == READY\_PROCESS\_SET--{Pro.ID} &&  WAITING\_PROCESS\_SET == WAITING\_PROCESS\_SET\-/{Pro.ID} &&  Pro.PROCESS\_STATE == WAITING  **END** |

### Arinc653抽象机实现

抽象机实现是根据低层抽象机定义的各功能模块形态，使用B\*语言，以基本数据类型、集合类型和元组类型为基础编写各模块的B\*程序。

依旧以SUSPEND函数为例，在给定了其B\*低层抽象机的基础上，根据其鲁棒性和功能性需求，使用B\*语言对SUSPEND具体抽象机进行实现，如表37所示。在抽象机实现的基础上，便可完成SUSPEND函数的形式验证及代码转换的工作。

表37 SUSPEND函数抽象机实现

|  |
| --- |
| **MACHINE** ProcessManagement  **OPERATIONS**  RETURN\_CODE\_TYPE **SUSPEND** (  **/\*in \*/**  PROCESS\_ID\_TYPE PROCESS\_ID)  {  **if**(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0  && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)  {  **return** INVALID\_MODE;  }  **if**(Pro.<ID:PROCESS\_ID> : PROCESS\_SET)  {  **if**(Pro.ID == CURRENT\_PROCESS.ID)  { |

表38 SUSPEND函数抽象机实现（续）

|  |
| --- |
| **return** INVALID\_PARAM;  }    **if**(Pro.PROCESS\_STATE == DORMANT)  {  **return** INVALID\_MODE;  }    **if**(Pro.PERIOD != INFINITE\_TIME\_VALUE)  {  **return** INVALID\_MODE;  }    **if**(Pro.PROCESS\_STATE == WAITING)  {  **return** NO\_ACTION;  }  **else**  {  Pro.PROCESS\_STATE = WAITING;  READY\_PROCESS\_SET = READY\_PROCESS\_SET -- {Pro.ID};  WAITING\_PROCESS\_SET = WAITING\_PROCESS\_SET \-/ {Pro.ID};  **return** NO\_ERROR;  }  }  **else**  {  **return** INVALID\_PARAM;  }  }  **END** |

## Arinc653内核验证

在6.2.3节给定的SUSPEND函数抽象机的基础上，使用本文实现的基于语境的形式化验证工具对其进行验证，验证序列及功能逻辑表达式如表39所示。因为SUSPEND函数中不包含循环结构程序，因此可由验证工具自动推导证明。将工具推导出的功能逻辑表达式与SUSPEND低级抽象机中的形式规约作对比，发现两者是完全等价的，因此SUSPEND的抽象机实现是正确的，即满足预期的功能性和鲁棒性需求，为代码转换的正确定奠定了基础。

表39 SUSPEND函数形式验证证明序列及功能逻辑表达式

|  |
| --- |
| **函数名CREATE\_SAMPLING\_PORT**  **证明序列是：**  **A1** = w(ERROR\_HANDLER\_PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0; PROCESS\_ID’0 == PROCESS\_ID’0; CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 == CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0){if(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)}  **A2** = CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0 => w(){return INVALID\_MODE}  **A3** = CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0 => CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A4** = ~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0) => w(PROCESS\_SET’0 == PROCESS\_SET’0; Pro’0 == Pro’0){if(Pro.<ID:PROCESS\_ID>:PROCESS\_SET)}  **A5** = (~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET) => w(CURRENT\_PROCESS.ID’0 == CURRENT\_PROCESS.ID’0; Pro.ID’0 == Pro.ID’0){if(Pro.ID==CURRENT\_PROCESS.ID)}  **A6** = ((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (Pro.ID’0 == CURRENT\_PROCESS.ID’0) => w(){return INVALID\_PARAM}  **A7** = ((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (Pro.ID’0 == CURRENT\_PROCESS.ID’0) => CREATE\_SAMPLING\_PORT == INVALID\_PARAM  **A8** = ((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0)) => w(Pro.PROCESS\_STATE’0 == Pro.PROCESS\_STATE’0){if(Pro.PROCESS\_STATE==DORMANT)}  **A9** = (((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (Pro.PROCESS\_STATE’0 == DORMANT) => w(){return INVALID\_MODE}  **A10** = (((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (Pro.PROCESS\_STATE’0 == DORMANT) => CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A11** = (((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT)) => w(INFINITE\_TIME\_VALUE == INFINITE\_TIME\_VALUE; Pro.PERIOD’0 == Pro.PERIOD’0){if(Pro.PERIOD!=INFINITE\_TIME\_VALUE)}  **A12** = ((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && |

表40 SUSPEND函数形式验证证明序列及功能逻辑表达式（续1）

|  |
| --- |
| (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (Pro.PERIOD’0 != INFINITE\_TIME\_VALUE) => w(){return INVALID\_MODE}  **A13** = ((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (Pro.PERIOD’0 != INFINITE\_TIME\_VALUE) => CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A14** = ((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE)) => w(Pro.PROCESS\_STATE == Pro.PROCESS\_STATE){if(Pro.PROCESS\_STATE==WAITING)}  **A15** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (Pro.PROCESS\_STATE’0 == WAITING) => w(){return NO\_ACTION}  **A16** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (Pro.PROCESS\_STATE’0 == WAITING) => CREATE\_SAMPLING\_PORT == NO\_ACTION  **A17** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => w(Pro.PROCESS\_STATE == WAITING){Pro.PROCESS\_STATE=WAITING}  **A18** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => w(READY\_PROCESS\_SET’0 == READY\_PROCESS\_SET’0; Pro.ID’0 == Pro.ID’0){READY\_PROCESS\_SET=READY\_PROCESS\_SET--{Pro.ID}}  **A19** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => READY\_PROCESS\_SET’1 == READY\_PROCESS\_SET’0--{Pro.ID} |

表41 SUSPEND函数形式验证证明序列及功能逻辑表达式（续2）

|  |
| --- |
| **A20** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 ==  ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE‘0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => w(WAITING\_PROCESS\_SET’0 == WAITING\_PROCESS\_SET’0; Pro.ID’0 == Pro.ID’0){WAITING\_PROCESS\_SET=WAITING\_PROCESS\_SET\-/{Pro.ID}}  **A21** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => WAITING\_PROCESS\_SET’1 == WAITING\_PROCESS\_SET’0\-/{Pro.ID}  **A22** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => w(){return NO\_ERROR}  **A23** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID’0 == CURRENT\_PROCESS.ID’0))) && (~(Pro.PROCESS\_STATE’0 == DORMANT))) && (~(Pro.PERIOD’0 != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE’0 == WAITING)) => CREATE\_SAMPLING\_PORT == NO\_ERROR && READY\_PROCESS\_SET’1 == READY\_PROCESS\_SET--{Pro.ID} && WAITING\_PROCESS\_SET’1 == WAITING\_PROCESS\_SET\-/{Pro.ID} && Pro.PROCESS\_STATE’1 == WAITING  **A24** = (~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (~(Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) => w(){return INVALID\_PARAM}  **A25** = (~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL’0 > 0 && PROCESS\_ID’0 == ERROR\_HANDLER\_PROCESS\_ID’0)) && (~(Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) => CREATE\_SAMPLING\_PORT == INVALID\_PARAM  **逻辑功能表达式是:**  **A3** = CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID => CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A7** = ((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (Pro.ID == CURRENT\_PROCESS.ID) => CREATE\_SAMPLING\_PORT == INVALID\_PARAM  **A10** = (((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID == CURRENT\_PROCESS.ID))) && (Pro.PROCESS\_STATE == DORMANT) => |

表42 SUSPEND函数形式验证证明序列及功能逻辑表达式（续3）

|  |
| --- |
| CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A13** = ((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID ==  ERROR\_HANDLER\_PROCESS\_ID)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID == CURRENT\_PROCESS.ID))) && (~(Pro.PROCESS\_STATE == DORMANT))) && (Pro.PERIOD != INFINITE\_TIME\_VALUE) => CREATE\_SAMPLING\_PORT == INVALID\_MODE  **A16** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID == CURRENT\_PROCESS.ID))) && (~(Pro.PROCESS\_STATE == DORMANT))) && (~(Pro.PERIOD != INFINITE\_TIME\_VALUE))) && (Pro.PROCESS\_STATE == WAITING) => CREATE\_SAMPLING\_PORT == NO\_ACTION  **A23** = (((((~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)) && (Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) && (~(Pro.ID == CURRENT\_PROCESS.ID))) && (~(Pro.PROCESS\_STATE == DORMANT))) && (~(Pro.PERIOD != INFINITE\_TIME\_VALUE))) && (~(Pro.PROCESS\_STATE == WAITING)) => CREATE\_SAMPLING\_PORT == NO\_ERROR && READY\_PROCESS\_SET == READY\_PROCESS\_SET--{Pro.ID} && WAITING\_PROCESS\_SET == WAITING\_PROCESS\_SET\-/{Pro.ID} && Pro.PROCESS\_STATE == WAITING  **A25** = (~(CURRENT\_PARTITION\_STATUS.LOCK\_LEVEL > 0 && PROCESS\_ID == ERROR\_HANDLER\_PROCESS\_ID)) && (~(Pro.<ID:PROCESS\_ID>:PROCESS\_SET)) => CREATE\_SAMPLING\_PORT == INVALID\_PARAM |

本课题组对ARINC653规范中分区管理、进程管理、分区间通信、时间管理、分区内通信和健康监控服务中的部分函数进行了B\*改写，统计49个B\*抽象机程序。本文使用基于语境的形式化验证工具对ARINC653规范中49个使用B\*描述的功能函数抽象机进行了形式化验证，其转换结果如表43至表53所示。

表43 分区管理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *GET\_PARTITION\_STATUS* | 得到分区状态 | 14 | 8 |

表44 进程管理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_PROCESS* | 创建进程 | 73 | 81 |
|  | *GET\_PROCESS\_ID* | 获取进程ID | 21 | 21 |
|  | *GET\_PROCESS\_STATUS* | 获取进程状态 | 35 | 24 |
|  | *SET\_PRIORITY* | 设置进程优先级 | 29 | 26 |
|  | *SUSPEND\_SELF* | 挂起当前进程 | 59 | 42 |
|  | *SUSPEND* | 挂起进程 | 42 | 33 |
|  | *RESUME* | 恢复进程 | 55 | 76 |
|  | *STOP\_SELF* | 停止当前进程 | 20 | 39 |
|  | *STOP* | 停止进程 | 40 | 68 |
|  | *LOCK\_PREEMPTION* | 增加分区锁优先级 | 24 | 23 |

表45 进程管理（续）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *ULOCK\_PREEMPTION* | 降低分区锁优先级 | 22 | 25 |
|  | *GET\_MY\_ID* | 获取当前进程ID | 19 | 12 |

表46 分区间通信采样端口服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_SAMPLING\_PORT* | 创建采样端口 | 82 | 90 |
|  | *WRITE\_SAMPLING\_MESSAGE* | 写采样端口消息 | 48 | 40 |
|  | *GET\_SAMPLING\_PORT\_ID* | 获取采样端口ID | 24 | 19 |
|  | *GET\_SAMPLING\_PORT\_STATUS* | 获取采样端口状态 | 26 | 22 |

表47 分区间通信队列端口服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_QUEUING\_PORT* | 创建队列端口 | 86 | 76 |
|  | *GET\_QUEUING\_PORT\_ID* | 获取队列端口ID | 24 | 18 |
|  | *GET\_QUEUING\_PORT\_STATUS* | 获取队列端口状态 | 26 | 18 |

表48 时间管理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *TIMED\_WAIT* | 时间等待 | 39 | 44 |
|  | *PERIODIC\_WAIT* | 周期等待 | 36 | 31 |
|  | *GET\_TIME* | 获取事件 | 14 | 8 |
|  | *REPLENISH* | 重新计时 | 37 | 29 |

表49 分区内通信缓冲区服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_BUFFER* | 创建缓冲区 | 59 | 67 |
|  | *SEND\_BUFFER* | 发送缓冲区消息 | 132 | 158 |
|  | *RECEIVE\_BUFFER* | 接收缓冲区消息 | 128 | 177 |
|  | *GET\_BUFFER\_ID* | 获取缓冲区ID | 23 | 24 |
|  | *GET\_BUFFER\_STATUS* | 获取缓冲区状态 | 26 | 24 |

表 50 分区内通信黑板服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_BLACKBOARD* | 创建黑板 | 42 | 38 |
|  | *DISPLAY\_BLACKBOARD* | 展示黑板消息 | 65 | 51 |
|  | *CLEAR\_BLACKBOARD* | 清空黑板消息 | 31 | 39 |
|  | *GET\_BLACKBOARD\_ID* | 获取黑板ID | 23 | 18 |
|  | *GET\_BLACKBOARD\_STATUS* | 获取黑板状态 | 25 | 24 |

表 51 分区内通信信号量服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_SEMAPHORE* | 创建信号量 | 58 | 42 |
|  | *WAIT\_SEMAPHORE* | 等待信号量 | 74 | 61 |
|  | *SIGNAL\_SEMAPHORE* | 释放信号量 | 47 | 59 |
|  | *GET\_SEMAPHORE\_ID* | 获取信号量ID | 23 | 18 |
|  | *GET\_SEMAPHORE\_STATUS* | 获取信号量状态 | 23 | 18 |

表 52 分区内通信事件服务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *CREATE\_EVENT* | 创建事件 | 38 | 28 |
|  | *SET\_EVENT* | 设置事件为UP | 45 | 43 |
|  | *RESET\_EVENT* | 设置事件为DOWN | 21 | 24 |
|  | *WAIT\_EVENT* | 等待事件 | 75 | 131 |
|  | *GET\_EVENT\_ID* | 获取事件ID | 24 | 16 |
|  | *GET\_EVENT\_STATUS* | 获取事件状态 | 25 | 16 |

表 53 健康监控

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数名称 | 功能描述 | B\*代码量 | 验证代码量 |
|  | *REPORT\_APPLICATION\_MESSAGE* | 报告应用信息 | 21 | 29 |
|  | *CREATE\_ERROR\_HANDLER* | 创建当前分区错误句柄 | 19 | 27 |
|  | *GET\_ERROR\_STATUS* | 获取错误状态信息 | 15 | 10 |
|  | *RAISE\_APPLICATION\_ERROR* | 抛出应用错误 | 28 | 42 |

现有较为成功的内核验证案例是澳大利亚国家信息通信技术研究所中心对L4内核使用Isabelle进行的形式化验证，整个L4内核包含约8700行C代码，验证代码量高达200000行，整个证明使用了约10人年。本课题组对ARINC653规范的49个函数使用B\*进行初步实现，内核代码量为4283行，使用基于语境的形式化验证工具对各抽象机进行验证，验证代码量为4197行，整个证明使用了月5人月。ARINC653内核验证与L4内核验证的对比如表54所示。

表54 ARINC653内核验证与L4内核验证比较

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 内核 | 应用 | 验证工具 | 内核代码量 | 转C代码量 | 验证代码量 | 人力 |
| *ARINC653* | 工业界 | 基于语境的形式化验证工具 | 4283 (B\*) | 7214 (C) | 4197 | 5人月 |
| *L4* | 学术界 | Isabelle/HOL | 8700 (C) | 8700 (C) | 200000 | 10人年 |

经对比，使用基于语境的形式化验证工具对ARINC653验证的验证代码量要远远小于L4内核的验证代码量，并且在花费的人力上也有较大优势，说明了本文设计并实现的基于语境的形式化验证工具具有真正的实用价值。

## 小结

本章基于课题组提出的B\*形式化方法及第五章设计并实现的基于语境的形式化验证工具，完成对了Arinc653规范的49个内核函数的建模与验证，建模代码量约为4300行B\*代码，验证代码约为4200行，耗时约为5人月。相对于澳大利亚国家信息通信技术研究所中心使用Isabelle对L4内核的验证，使用基于语境的形式化验证工具对Arinc653内核验证在效率及人力上都占有一定优势，说明了其具有真正的实用价值。

使用B\*语言构造Arinc653内核函数的抽象机，并对抽象机进行形式化验证和代码转换，便可得到可执行程序。因为设计层的抽象机经过了形式验证，因此是符合Arinc653内核函数低层抽象机形式化需求的，即实现的抽象机是符合Arinc653规范的。对经过验证的内核函数抽象机中的集合结构等使用具体的数据结构如链表、树等进行替换，便可完成代码的转换（其正确性使用测试和验证共同保证），从而得到最终的可执行程序。

# 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

B\*语言的关键字如表55所示，关键字不能作为标识符出现。

表55 B\*语言的关键字

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MACHINE | INCLUDE | ATTRIBUTES | OPERATIONS | ROBUST | END |
| FUNCTION | Precondition | Postcondition | const | unsigned | char |
| short | int | long | float | double | string |
| propossition | set | enum | struct | Nil | true |
| false | void | return | if | else | while |
| typedef |  |  |  |  |  |

B\*语言的词法如表56所示：

表56 B\*词法规则

|  |  |  |
| --- | --- | --- |
|  | *MACHINE* ::= MACHINE | 抽象机定义开始 |
|  | *ATTRIBUTES*::=ATTRIBUTES | 变量声明 |
|  | *DEFINE* :: = #define | 宏定义 |
|  | *INCLUDE* ::= INCLUDE | 包含其他抽象机 |
|  | *ROBUST*::=ROBUST | 鲁棒性声明 |
|  | *OPERATIONS* ::= OPERATIONS | 函数声明 |
|  | *END* ::= END | 表示整个程序结束 |
|  | *FUNCTION* ::= FUNCTION | 功能性声明 |
|  | *PRECONDITION* ::= Precondition | 前置条件声明 |
|  | *POSTCONDITION* ::= Postcondition | 后置条件声明 |
|  | *CONST* ::= const | 不可变类型关键字 |
|  | *UNSIGNED* ::= unsigned | 无符号类型关键字 |
|  | *CHAR* ::= char | 字符类型 |
|  | *SHORT* ::= short | 短整型类型 |
|  | *INT* ::= int | 整数类型 |
|  | *LONG* ::= long | 长整型类型 |
|  | *DOUBLE* ::= double | 双精度浮点数类型 |
|  | *FLOAT* ::= float | 单精度浮点数类型 |
|  | *STRING* ::= string | 字符串类型 |
|  | *PROPOSITION* ::= proposition | 命题类型 |
|  | *SET* ::= set | 集合类型 |
|  | *STRUCT* ::= struct | 元组类型 |
|  | *NIL* ::= nil | 空 |
|  | *TRUE* ::= true | 真 |
|  | *FALSE* ::= false | 假 |
|  | *RETURN* ::= return | 返回语句 |

表57 B\*词法规则（续1）

|  |  |  |
| --- | --- | --- |
|  | *WHILE* ::= while | 循环语句 |
|  | *IF* ::= if | 条件语句if分支 |
|  | *ELSE* ::= else | 条件语句else分支 |
|  | *ID* ::= [\_A-Za-z][\w\_]\* | 标识符 |
|  | *CHAR\_L* :: = \'.\' | 字符字面量 |
|  | *STRING\_L* :: = \".\*\" | 字符串字面量 |
|  | *LPAREN* :: = ( | 左小括号 |
|  | *RPAREN* :: = ) | 右小括号 |
|  | *LBRACE* :: = { | 左大括号 |
|  | *RBRACE* :: = } | 右大括号 |
|  | *LBRACKET* :: = [ | 左中括号 |
|  | *RBRACKET* :: = [ | 右中括号 |
|  | *SEMICOLON* :: = ; | 分号 |
|  | *COMMA* :: = , | 逗号 |
|  | *POINT* :: = . | 点 |
|  | *PLUS* :: = + | 加号 |
|  | *SETPLUS* :: = ++ | 集合中添加元素符号 |
|  | *MINUS* :: = - | 减号 |
|  | *UINT* :: = [1-9](0-9)\* | 0 | 无符号整型字面量 |
|  | *UFLOAT* ::= \d+\.\d+ | 字面量“浮点数” |
|  | *MULTIPLY* ::= \* | 乘号 |
|  | *DIVIDE* ::= / | 除号 |
|  | *REM* ::= % | 取模 |
|  | *NOT* ::= ~ | 非 |
|  | *CONJUNCTION* ::= && | 合取 |
|  | *DISJUNCTION* ::= || | 析取 |
|  | *ADDR* ::= & | 取地址 |
|  | *IMPLICATION* ::= => | 蕴含 |
|  | *BELONG* ::= : | 属于 |
|  | *NOTBELONG* ::= !: | 不属于 |
|  | *FORALL* ::= # | 任意 |
|  | *EXIST* ::= ! | 存在 |
|  | *EQ* ::= == | 等于 |
|  | *LE* ::= <= | 小于等于 |
|  | *LT* ::= < | 小于 |
|  | *NE* ::= != | 不等于 |
|  | *GE* ::= >= | 大于等于 |
|  | *GT* ::= > | 大于 |
|  | *SEPERATOR* ::= | | 按位或 |
|  | *INTERSECTION* ::= /-\ | 集合交运算符 |
|  | *UNION* ::= \-/ | 集合并运算符 |
|  | *DIFFERENCE* ::= -- | 集合差运算符 |
|  | *ASSIGNMENT* ::= = | 赋值运算符 |

表58 B\*词法规则（续2）

|  |  |  |
| --- | --- | --- |
|  | *ADDRGET* ::= -> | 取元组指针中的元素 |

B\*语言的程序结构如表59、表60和表61所示：

表59 高层抽象机语法结构

|  |  |  |  |
| --- | --- | --- | --- |
|  | *MachineH* | ::= | *MACHINE ID*  [*Includes*]  [ *Attributes*]  [*DeclareOperations*]  *END* |

表60 高层抽象机语法结构

|  |  |  |  |
| --- | --- | --- | --- |
|  | *MachineH* | ::= | *MACHINE ID*  [*Includes*]  [ *Attributes*]  [*CDeclareOperations*]  *END* |

表61 抽象机语法结构

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Machine* | ::= | *MACHINE ID*  [*Includes*]  [ *Attributes*]  [*Operations*]  *END* |

类似于B语言，MACHINE…END代码块被称作一个抽象机，B\*语言开发系统主要有三层，分为高层抽象机、低层抽象机和具体抽象机。高层抽象机描述系统，即系统“有什么”，是系统的初步设计；低层抽象机详细定义软件的功能，该层相当于详细的形式化需求文档，描述系统“做什么”；具体抽象机是系统的设计实现。抽象机是软件或系统某个功能模块的抽象模型，是软件的抽象、形式化的描述。

B\*语言用包含语句来组织抽象机之间关系，声明和定义语句包含变量、常量的声明和定义，操作的声明和定义，函数鲁棒性和功能性的定义等，具体如表62所示：

表62 B\*语言包含、声明和定义

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Includes* | ::= | ( *INCLUDE* *STRING\_L* *SEMICOLON* )+ |
|  | *Attributes* | ::= | *ATTRIBUTE* [*Defines*] *CVDefine* |
|  | *Defines* | ::= | ( *DEFINE ID Element* )+ |
|  | *CVDefines* | ::= | *VarDefine |*  *CstDefines* |
|  | *VarDefine* | ::= | ( *Type PointIdDefine [ASSIGNMENT Element]* ) *+ COMMA |*  *TYPE* ( *EnumType ID | Type PointIdDefine | TupleType PointIdDefine)* |

表63 B\*语言包含、声明和定义（续）

|  |  |  |  |
| --- | --- | --- | --- |
|  | *DeclareOperations* | ::= | *OPERATIONS Declares* |
|  | *Declares* | ::= | ( *FunctionDeclare* )+ |
|  | *CDeclareOperations* | *::=* | *OPERATIONS* ( *CDeclares* )+ |
|  | *CDeclares* | *::=* | *FunctionDeclare Robust Function* |
|  | *FunctionDeclare* | *::=* | *Type PointIdDefine*  *LPAREN ParameterDefineList RPAREN SEMICOLOM* |
|  | *ParameterDefineList* | *::=* | ( *Type PointIdDefine* )+COMMA |
|  | *Robust* | *::=* | *ROBUST* ( *PRECONDITION ProExp POSTCONDITION* )+ |
|  | *Function* | *::=* | *FUNCTION* ( *PRECONDITION ProExp POSTCONDITION* )+ |
|  | *Operations* | ::= | *OPERATIONS* ( *FunctionDefine* )+ |
|  | *FunctionDefine* | ::= | *Type PointIdDefine LPAREN ParameterDefineList RPAREN*  *LBRACE ComStatement* |

B\*语言的语句主要有空语句、函数调用语句、赋值语句、返回语句、while循环语句、if语句、语句块，在这些传统的语句基础上，增加了集合相关的语句，主要有向集合中增加元素语句、从集合中减去元素语句和从集合中按不同索引取元素语句。B\*语句的语法结构如表64所示。

表64 B\*语言包含、声明和定义

|  |  |  |  |
| --- | --- | --- | --- |
|  | *ComStatement* | ::= | [ *Attributes* ] ( *Statement* )\* |
|  | *Statement* | ::= | *SEMICOLON* |  *FunctionCall |*  *SetAddElement |*  *SetMinusElement |*  *Assignment |*  *ReturnStatement |*  *WhileStatement |*  *IfStatement |*  *LBRACE* ( *Statement* )\* *RBRACE* |
|  | *FunctionCall* | ::= | *ID LPAREN ParameterValueList RPAREN* |
|  | *ParameterValueList* | ::= | ( *Element* )+COMMA |
|  | *Assignment* | ::= | *PointIdExp* ( *POINT ID | ADDRGET ID* )*\**  *ASSIGNMENT Element SEMICOLON* |
|  | *ReturnStatement* | ::= | *RETURN* [ *Element* ] SEMICOLON |
|  | *WhileStatement* | ::= | *WHILE LPAREN ProExp RPAREN Statement* |
|  | *IfStatement* | ::= | *IF LPAREN ProExp RPAREN Statement* [ *ELSE Statement* ] |
|  | *SetAddElement* | ::= | *ID SETPLUS Element* |
|  | *SetMinusElement* | ::= | *ID DIFFERENCE Element* |

B\*的表达式除了基本的数学表达式还有命题表达式（推出命题、与命题、或命题、存在命题、任意命题）、条件表达式（用在条件判断中，是明天表达式的子集）和集合表达式（集合与集合的并、差、交操作，集合与元素的属于、不属于操作）。B\*表达式的语法如表65所示。

表65 B\*语言表达式文法

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Expression* | ::= | *ProExp* |
|  | *ProExp* | ::= | *OrProExp* ( *IMPLICATION OrProExp* )\* |
|  | *ImpProExp* | ::= | *OrProExp IMPLICATION OrProExp* |
|  | *OrProExp* | ::= | *AndProExp* ( *DISJUNCTION AndProExp* ) |
|  | *AndProExp* | ::= | *ProTermExp* ( *CONJUNCTION ProTermExp* ) |
|  | *ProTermExp* | ::= | *QuanExp |*  *ConditionExp* |
|  | *QuanExp* | ::= | *EXIST IdList LPAREN ProExp RPAREN |*  *FORMAL IdList LPAREN ImpProExp RPAREN* |
|  | *IdList* | ::= | *ID* ( *COMMA ID* )*\* |*  *LPAREN ID* ( *COMMA ID* )*\* RPAREN* |
|  | *ConditionExp* | ::= | *MathSetExp* [(  *GT MathSetExp |*  *NOTBELONG MathSetExp |*  *LE MathSetExp |*  *LT MathSetExp |*  *GE MathSetExp |*  *BELONG MathSetExp |*  *EQ MathSetExp |*  *NE MathSetExp*  )] |
|  | *MathSetExp* | ::= | *MathSetExp0* (  *PLUS MathSetExp0 |*  *MINUS MathSetExp0 |*  *DIFFERENCE MathSetExp0*  ) |
|  | *MathSetExp0* | ::= | *UnaryExp* (  *MULTIPY UnaryExp |*  *DIVIDE UnaryExp |*  *REM UnaryExp |*  *INTERSECTION UnaryExp |*  *UNION UnaryExp*  )*\** |
|  | *UnaryExp* | ::= | ( *PLUS | MINUS* ) *Term |*  *UnaryExp0* |
|  | *UnaryExp0* | ::= | *NOT UnaryExp0|*  *MULTIPLY UnaryExp0 |*  *ADDR UnaryExp0* |

表66 B\*语言表达式文法（续）

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Term* | ::= | (  *NIL |*  *STRING\_L |*  *TRUE |*  *FALSE |*  *UINT |*  *UFLOAT |*  *CHAR\_L |*  *LBRACE* ( *Element* )+ *COMM RBRACE |*  *LT* ( *Element* )*+COMMA GT |*  *FunctionCall |*  *ID |*  *LPAREN Element RPAREN*  )(  *POINT LT* ( *ID | UINT* ) *BELONG Element* ( *COMMA ID* ) |  *POINT ID |*  *ADDRGET ID*  )*\** |
|  | *Element* | ::= | *Expression* |

B\*主要有基本数据结构、枚举、元组和集合。类型相关的文法如表67所示：

表67 B\*类型

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Type* | ::= | *NormalType |*  *SetType |*  *Id* |
|  | *SetType* | ::= | *SET LT Type GT* |
|  | *TupleType* | ::= | *STRUCT*  *LBRACE* ( *Type PointIdDefine SEMICOLON* )\* *RBRACE* |
|  | *EnumType* | ::= | *ENUM*  *LBRACE* ( *ID* [ *ASSIGMENT UnaryExp* ])*+COMMA RBRACE* |
|  | *NormalType* | ::= | *CHAR |*  *SHORT |*  *INT |*  *LONG |*  *FLOAT |*  *DOUBLE |*  *STRING |*  *PROPOSITION |*  *VOID |*  *UNSIGNED SHORT |*  *UNSIGNED INT |*  *UNSIGNED LONG* |

# 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。