

中图分类号：TP311
学科分类号：085211

论文编号：1028716 16-SZ010

硕士学位论文

C 程序运行时监控和验证的插桩方法 研究与应用

研究生姓名	朱云龙
专业类别	工程硕士
专业领域	计算机技术
指导教师	李绪蓉 副教授（陈哲 副教授）

南京航空航天大学

研究生院 计算机科学与技术学院

二〇一六年三月

Nanjing University of Aeronautics and Astronautics

The Graduate School

College of Computer Science and Technology

Research on Instrumentation Methods in Runtime Monitoring and Verification for C Programs and Applications

A Thesis in

Computer Technology

by

Zhu Yunlong

Advised by

Associate Prof. Li Xurong (Associate Prof. Chen Zhe)

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Engineering

March, 2016

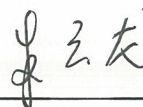
承诺书

本人声明所呈交的硕士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：



日

期：

2016.3.29

摘 要

随着软件在社会生活应用中的不断深入,软件系统的体积和复杂度都呈现出迅速增长的趋势,软件可靠性问题也相应成为软件行业发展不容忽视的重要方面。在现有的软件验证技术中,运行时验证作为一种轻量级的验证技术,综合了运行时监控技术和形式化规约技术,是当前软件验证领域的一个研究热点问题。而在运行时验证的技术实现中,一方面需要实现待验证性质的描述和相应验证器的生成,另一方面需要将监控器和验证器集成到待监控的软件系统中,即监控和验证代码的插桩。由于软件体积和复杂度的不断增加,基于日志 API 监控或手动进行监控验证代码插桩明显不能满足不同平台,不同类型系统监控验证的需求。随着编译器技术和面向方面编程技术的不断发展,利用编译器实现代码自动插桩,借助面向方面语言实现插桩位置描述和模块化管理,正在成为推动运行时验证实际应用的重要研究领域。

本文分析了 C 语言的编译流程,LLVM 开源编译器框架及其 Clang 编译器前端,在此基础上参照面向方面编程的概念,设计并实现了一种针对 C 程序的面向方面语言 MOVEC,其相应编译器能够解析监控器插桩位置的描述并利用源代码插桩技术实现插桩需求。同样基于源代码插桩的技术,本文设计并实现了软件中除 0,整数溢出和变量使用前未初始化错误验证器的插桩。通过将 MOVEC 语言编译器和现有针对 C/C++程序的面向方面语言编译器 AspectC,AspectC++在准确性,插桩时间,程序性能影响和程序代码膨胀等方面进行对比,表明 MOVEC 语言能够在保证对程序性能的低影响前提下,实现更高的准确性,更低的插桩时间,更小的代码膨胀率。实验同时验证了三种常见错误验证器设计和插桩方法的正确性。本文的工作为运行时验证监控器和验证器插桩的自动化,高效率,跨平台实现提供了基础,能够促进运行时验证在实际生活中的应用。

关键词: 运行时监控, 运行时验证, 源代码插桩, C 语言, 面向方面编程

ABSTRACT

With more and more software is applied in social life, the size and complexity of software systems are increasing rapidly, software reliability becomes an important aspect of software development accordingly. Among the exist software verification technologies, as a lightweight software verification method, runtime verification combines the technology of runtime monitoring and the techonolgy of formal specification, so it is rising as a pop research area. The key to runtime verification has two aspects, one for the description of property to be checked and generation of validator, the other one for the instrumentation of monitor and validator. Owing to the increasing size and complexity of software systems, monitoring based on log API or manual code instrumentation is not applicable for the verification of systems from different platforms and different areas. With the development of compiler technology and aspect-oriented programming technology, using compiler to realize the automated instrumentation and aspect-oriented language to describe and modularly manage the instrumentation location is becoming an important research filed for the practical application of runtime verification.

This thesis analyzes the compilation process of C programs, LLVM open source compiler framework and its Clang frontend. Based on these tools and the concept of AOP, we design and implement an AOP language MOVEC for C programs, with which monitor instrumentation location's description can be analyzed and realized automatically using source code instrumentation. We also design and implement a validator instrumentation tool for errors such as dividing by zero, integer overflow and value uninitialization which is also based on source code instrumentation. The final experiment of comparison between MOVEC, AspectC, AspectC++ shows its advantages in higher accuracy, less instrumentation time consumption and less code bloat with no more running time overhead. Another one experiment shows the validity of validator design and instrumentation algorithm. Work in this thesis can be a foundation for the automated, efficient, cross-platform instrumentation for monitor and validator in runtime verification, and promote the practical application of runtime verification.

Keywords: runtime monitoring, runtime verification, source code instrumentation, C language, aspect-oriented programming

目 录

第一章 绪论	1
1.1 课题研究背景及意义	1
1.2 当前研究现状及选题依据	2
1.3 论文研究内容	5
第二章 编译知识和相关工具的介绍	6
2.1 语言的编译流程	6
2.2 LLVM 编译器框架	6
2.3 Clang 前端和抽象语法树	7
2.4 Flex 词法分析生成器和 Bison 语法分析生成器	10
2.5 本章小结	12
第三章 面向方面语言	13
3.1 传统关注点分离模型	13
3.1.1 自适应面向对象编程	13
3.1.2 面向主题编程	13
3.1.3 元对象编程	14
3.1.4 组合过滤编程	15
3.2 面向方面编程的基本概念	15
3.3 面向方面编程的技术关键	16
3.3.1 接入点设计	16
3.3.2 方面表达设计	17
3.3.3 切入点匹配设计	18
3.3.4 通知调用织入设计	18
3.4 针对 C 的面向方面解决方案对比	19
3.4.1 AspectC	19
3.4.2 AspectC++	19
3.4.3 InterAspect	20
3.5 本章小结	20
第四章 MOVEC 语言设计与实现	21
4.1 MOVEC 语言的语法设计	21

4.1.1 切入点	21
4.1.2 通知	23
4.1.3 方面	24
4.2 MOVEC 语言编译器的设计与实现	25
4.2.1 方面代码解析	26
4.2.2 词法分析与宏替换	26
4.2.3 语法分析与切入点的匹配	29
4.2.4 代码转换	32
4.2.5 单文件编译与项目编译	38
4.3 本章小结	38
第五章 C 语言常见错误的运行时验证插桩	40
5.1 常见软件错误定义及相关研究	40
5.1.1 C 语言及其常见错误定义	40
5.1.2 整数溢出错误验证插桩相关研究	41
5.2 常见错误的验证插桩设计与实现	42
5.2.1 除 0 错误的验证插桩设计与实现	42
5.2.2 整数溢出的验证插桩设计与实现	43
5.2.3 未初始化变量引用的验证插桩设计与实现	47
5.3 本章小结	49
第六章 实验与分析	50
6.1 实验目标系统介绍	50
6.2 实验软硬件平台介绍	51
6.3 实验结果及分析	51
6.3.1 MOVEC 编译器实验	51
6.3.2 常见软件错误实验	55
6.4 本章小结	56
第七章 总结与展望	57
7.1 论文总结	57
7.2 工作展望	57
参考文献	59
致 谢	64
在学期间的研究成果及发表的学术论文	65

图表清单

图 2.1 编译器编译流程.....	6
图 2.2 Clang 中声明类继承结构.....	8
图 2.3 Clang 中语句类继承结构.....	9
图 2.4 Clang 中类型类继承结构.....	9
图 2.5 ClangTool 执行流程.....	10
图 2.6 flex 输入文件示例.....	11
图 2.7 bison 输入文件示例.....	11
图 4.1 通知语法结构.....	23
图 4.2 参数化通知.....	23
图 4.3 MOVEC 方面示例.....	25
图 4.4 MOVEC 编译器组成结构.....	25
图 4.5 方面数据结构.....	26
图 4.6 切入点继承结构.....	27
图 4.7 宏替换算法.....	28
图 4.8 复合切入点搜索及匹配算法.....	32
图 4.9 接入点上下文数据结构.....	33
图 4.10 call && callp 代码转换规则.....	35
图 4.11 inexec 动态判定条件生成算法.....	35
图 4.12 execution 代码转换规则.....	36
图 4.13 变量赋值取值与函数调用等价转换.....	37
图 4.14 多通知匹配代码逻辑.....	37
图 5.1 除 0 验证代码转换规则.....	43
图 5.2 除 0 验证判断函数.....	43
图 5.3 一元操作代码转换规则.....	44
图 5.4 一元操作溢出验证判断函数.....	45
图 5.5 简单二元操作代码转换规则.....	45
图 5.6 简单二元操作溢出验证判断函数.....	46
图 5.7 复合赋值二元操作代码转换规则.....	46
图 5.8 类型转换代码转换规则.....	47
图 5.9 未初始化变量引用验证算法.....	48
图 5.10 变量赋值及变量引用代码转换规则.....	49

图 6.1 验证插桩代码实例.....	56
表 4.1 静态切点关键字.....	21
表 4.2 动态切入点关键字.....	22
表 4.3 接入点上下文信息.....	24
表 4.4 切入点与语法树节点映射关系.....	30
表 6.1 测试软件平台信息.....	51
表 6.2 实验 ACC、AC++、MOVEC 插桩时间，正确性及目标程序效率变化	53
表 6.3 实验 ACC、AC++、MOVEC 插桩后代码膨胀率	54
表 6.4 常见错误验证时间和正确性数据.....	55

注释表

缩略词	英文全称
AOP	Aspect-oriented Programming
POP	Process-oriented Programming
OOP	Object-oriented Programming
AST	Abstract Syntax Tree
LLVM	Low Level Virtual Machine
SDK	Software Development Kit
IOC	Inversion of Control
ACC	Aspect-oriented C
AC++	Aspect-oriented C++
JSON	JavaScript Object Notation
FLEX	Fast Lexical Analyzer
RV	Runtime Verification
NVD	National Vulnerability Database

第一章 绪论

1.1 课题研究背景及意义

现代生活中,软件正以前所未有的速度融入到日常生活的方方面面。软件已经成为整个社会运行不可缺少的一部分,银行,通信,教育,交通,乃至核工业等都在利用软件助力行业和社会的发展。然而,近年来软件系统的大小和复杂度呈现出迅速增长的态势,而且这种趋势在未来依然会持续。来自产业界的数据显示,各种系统和应用的大小在过去的 40 年里是以指数级别速度增长的^[1]。随着这种对软件依赖性的逐年增加,软件中的失效可能会给商业系统,尤其是安全关键系统带来巨大损失甚至致命的后果,例如,存在于 Adobe PDF, Flash 和 Microsoft Office 软件中的 0-day 漏洞导致一系列的黑客攻击^[2],存在于控制软件中的整数溢出漏洞导致阿丽亚娜 5 型运载火箭在 1996 的首次鉴定发射时遭遇失败^[3]。因此,软件的可靠性正在成为软件行业发展中不可忽视的一个重要方面。

软件可靠性是指在特定环境和特定时间段内软件保持运行不失效的概率,而软件可靠性工程是指利用一些工程上的技术来开发和维护软件系统,使得这些系统的可靠性能进行量化的评价^[4]。Lyu 按照错误产生的周期概括了提高软件系统可靠性的四个主要方面:通过规范开发进行错误预防和错误避免;通过验证和确认技术实现错误检测和移除;通过冗余设计使系统服务在错误发生情况下依然满足要求;通过软件可靠性模型等对错误存在性,未来失效发生可能性和失效影响等进行预测^[5]。其中软件验证技术一直是研究的重要方面,现有的软件验证技术主要包括如下几种,静态分析^[6],模型检测^[7],软件测试^[8]。静态分析是指通过类型推断,数据流分析和约束分析等找出可能存在的错误,优点是可以在系统开发早期发现错误,但同时也会带来大量的误报和漏报;模型检测通过对软件结构的分析,将系统转化成有限状态空间模型,并利用算法验证其完备性,优点是模型检测是可证明的,缺点是形式化验证会带来状态空间爆炸问题,验证需要时间较长且模型的行为和系统的实际行为之间会存在一定的差距;软件测试是指通过设计测试用例,并在此基础上运行程序,观察程序的行为和结果,优点是实现了测试用例管理和执行的自动化,缺点是测试用例无法实现全覆盖,错误会出现遗漏,另外验证性质的描述采用的是直接定义的方式,无法通过形式化表达自动生成,限制了验证的能力。

区别于前述四种主要在软件开发过程中应用的验证技术,运行时验证是一种在软件实际运行阶段进行性质分析验证的技术,作为一种新近提出的轻量级验证技术,能够实现对软件运行的实时监控,验证软件是否符合或违反给定的正确性属性,对提高软件的可靠性有重要的意义^[9]。运行时验证是运行时监控技术和形式化规约技术的结合体,其通过监控软件运行过程获取实时状态,避免了模型检测中形式化验证带来的状态空间爆炸和复杂度问题,并考虑到了程序

实际运行过程的环境等影响因素，同时运行时验证采用形式化语言对待验证性质进行描述，例如线性时序逻辑，正则表达式，自动机等，使运行时验证对性质表达更加丰富和灵活。运行时验证在性质分析验证的基础上，往往还支持对程序运行流程的控制，当程序运行进入某种错误状态，可以引导程序进入安全状态或者直接终止程序的运行，避免程序的错误运行带来更大的损失或者严重的事故。目前运行时验证技术已经被广泛应用在硬件运行控制^[10]，Web 应用^[11]和 Android 应用的监控^[12]，网络^[13]等各个领域。

在运行时验证实现的关键技术中，主要包括两个部分，一个是待验证性质的描述和相应验证器的生成，二是监控器和验证器的系统集成。在性质描述方面，运行时验证采用了和模型检测相类似的技术，通过线性时序逻辑^[14]，参数化命题^[15]，正则表达式^[16]，自动机^[17]等逻辑语言进行描述。在验证器生成方面，验证器是指一类特殊的组件，可以读取一个有限事件序列，并给出一个特定的结论，验证器依据用户所指定的性质自动生成，并随着对应的执行代码集成到待监控的系统当中，目前验证器主要采用基于自动机的验证算法和基于公式重写的验证算法生成^[9]。由于在性质描述和验证器生成方面，部分继承了模型检测中的内容，目前已有大量的成熟理论或算法可供参考，具有普遍的语言和算法通用性；而在监控器和验证器的系统集成方面，对应不同的开发平台，不同的目标语言，不同的性质描述规范会带来集成方法的不同，集成方法的设计反过来又会影响运行时验证本身性质的表达能力。因此，如何设计并实现运行时验证监控器和验证器的高效系统集成，成为运行时验证领域研究的一个重要和热点的方面，对于推动运行时验证在实际系统中的应用具有重要价值。

1.2 当前研究现状及选题依据

针对运行时验证如何实现监控器和验证器的系统集成，主要包含如下四种方法，一是采用代码插桩，其中代码可以是源代码，字节码或者二进制代码，代码插桩的方式可以使监控器，验证器和待验证系统之间建立起紧密的联系；二是通过特定的日志框架进行监控，日志信息记录了系统运行过程中的相关信息，提供给验证器进行分析验证；三是利用特定追踪软件或追踪硬件来获取系统运行过程中的相关信息实现监控，例如 Unix 中 `strace` 工具可以追踪进程执行时的系统调用和所接收的信号，利用 BusMOP 框架可以设计硬件来追踪系统总线上的所有交互，并最终发送给验证器达成系统性质的验证^[18]。上述三种集成方法中，应用最广泛的是代码插桩，相比另外两种集成方式，通过对源代码，字节码或二进制代码的程序结构分析，可以使获得的系统信息更加准确，种类更加丰富，从而提高整个运行时框架的验证能力，日志框架或追踪软件和追踪硬件从不同程度上限制了获取系统运行状态的能力，而且代码插桩方式能直接应用于在线运行时验证的设计中。

随着运行时验证研究的不断推进，目前已经发展出大量的运行时验证框架，例如 Jass^[19]，

JavaMOP^[20], JavaMaC^[21], JavaPathExplorer^[22], TraceMatches^[16], TraceContract^[23], Eagle^[24], RulerR^[25], J-LO^[15]等。其中, Jass 采用了基于注释的代码生成技术, JavaMac, JavaPathExplorers, EAGLE 和 RulerR 采用了字节码插桩的技术, TraceMatches, J-LO 和 JavaMOP 综合运用了面向方面编程技术和字节码插桩, 可见代码插桩是运行时验证监控器和验证器集成的普遍选择。

JavaMOP 除了代码插桩技术之外还采用了面向方面编程技术, 这是为了实现对监控器插桩位置自定义和模块化管理而加入的技术。因为待插桩的监控器可以理解为软件工程中的关注点, 所以监控器的集成问题与面向方面编程乃至整个软件开发方法研究所要解决的关注点分离问题类似。从软件开发方法发展的角度, 面向方面编程 (Aspect-oriented Programming, 简称 AOP) 是在面向过程和 (Process-oriented Programming, 简称 POP) 和面向对象编程 (Object-oriented Programming, 简称 OOP) 的基础上提出的一种对于核心关注点和横切关注点的分离和整合方法。

面向过程的编程方法最早由 Yourdon 和 Constantine 提出, 在软件开发的初期应用十分广泛^[26]。软件开发人员根据功能的不同, 将系统划分为不同的模块, 通过相应的程序代码来定义数据, 并构造函数以实现这些模块, 也就是软件开发中描述的数据结构和算法的概念。虽然面向过程的编程方法并不符合实体的直观感受, 但在驱动开发, 底层协议开发等方面有其特殊的优势。

自 20 世纪 90 年代以来, 面向对象编程方法的出现, 使得软件开发人员对系统功能的理解更加接近于现实生活中的实体^[27]。每个模块就相当于一台机器, 模块中的数据相当于机器中零部件, 而模块中的方法相当于机器对外提供的功能。数据被隐藏于模块之中, 通过方法对其他模块提供服务。面向对象的编程方法和面向过程的编程方法的区别主要体现在: 1) 通过抽象实现了对类的数据和行为的区分和提取; 2) 通过封装实现了类结构中部分数据和行为的隐藏; 3) 继承体现了从一般到特殊的过程, 可以在既有类的结构上直接进行数据或者行为的扩展, 而不需要重复的代码; 4) 多态是指继承所产生的类之间可能有同样名称的行为, 但由于不同的类对该行为的理解的不同, 会导致在实际运行过程中的表现也有所不同。

虽然面向对象编程以一种抽象的形式, 实现了系统中层次关注点的封装, 但依然无法解决各模块之间调用关系复杂, 代码重复率高的问题, 而这些问题产生的根本原因在于系统中存在的跨模块控制行为, 如权限控制, 消息记录等, 即一种特殊的关注点, 横切关注点。

1997 年, Gregor 等人在欧洲面向对象编程大会上提出了面向方面编程的概念, 并在之后的时间里得到广泛的关注和发展^[28]。尤其是 2001 年, Gregor 所在团队发布的针对 Java 的面向方面语言 AspectJ, 为面向方面编程在实际系统中的应用提供了助力, 促进了面向编程相关领域的研究。面向方面编程通过将关注点以方面的形式进行模块化存储, 并利用相应的编织方法将关注点整合到目标系统中, 最终实现关注点的高度分离和重用。

面向方面编程思想可以使软件开发中的核心业务模块和横切模块实现了高度的分离和透明,横切模块开发人员只需要注重于横切模块内部的设计和横切位置的定义,原有的核心业务模块不需要做任何的修改。这种特点,使得面向方面编程在监控器遍布系统各个模块的软件运行时验证中非常适用,也成为软件运行时验证领域代码插桩普遍采用的技术。下面将就其中应用较为广泛,有行业影响力的工具,从语法表达和应用场景角度进行简单的介绍。

AspectJ^[29]是施乐公司 AOP 小组于 2001 年发布的针对 Java 语言的面向方面扩展,目前由 Eclipse 基金会作为一个开源项目进行维护和开发,包含有独立的开发工具和对 Eclipse 集成开发环境的支持,目前已更新到 1.8.7 版本,是目前使用最广泛的面向方面语言。AspectJ 是对 Java 语法和语义的扩展,定义了一系列方面关键字,利用这些关键字实现方面的声明。

AspectWerkz^[30]是由 Jonas Boner 和 Alex Vasserur 开发出的一个动态,轻量级和高效的针对 Java 的面向方面编程框架。AspectWerkz 采用普通 Java 类的形式进行方面描述,不改变原有的 Java 语法,通过 Java 注释或者 XML 文件的方式实现 Java 语义的扩展。目前 AspectWerkz 已经和 AspectJ 完成了功能合并,成为了 AspectJ 的一部分。

JBossAOP^[31]是 JBoss 公司在 JBoss4 中引入的对面向方面编程的支持,极大的缩短了软件的开发周期。目前 JBossAOP 作为一种针对 Java 的面向方面框架,既可以在 JBoss 服务器中使用,也适用于其他编程环境。JBossAOP 主要采用 XML 文件来完成,同时也支持注释的声明方式。

SpringAOP^[32]是 Spring 框架的一个重要组成部分,不同于其他 AOP 语言,SpringAOP 设计的主要目的是配合 Spring 的 IOC 控制反转功能,解决企业级应用中的常见问题,所以目前只支持对方法执行的切入。在使用方面,SpringAOP 完全使用 XML 实现对方面的声明,和 Spring 使用 XML 文件进行组件应用的配置是一致的。

从上述应用工具列表中可以看出,目前大部分面向方面语言的研究和应用集中在对 Java 语言的支持上。针对其他语言的 AOP 支持仍处于研究的起步阶段,例如针对 C/C++ 语言面向方面支持的 AspectC^[33]和 AspectC++^[34],针对 .net 语言面向方面支持的 Sourceweave.NET^[35],针对 Matlab 语言面向方面支持的 AspectMatlab^[36],从功能,适用性等角度都还没有达到 Java 语言可以实现商用的级别。国内对于面向方面编程的研究主要集中在 AOP 技术的理解与应用,例如利用 AOP 进行日志监控,增强云服务可信性^[37],利用基于 AOP 的过程更新模型,实现网络业务过程的动态更新^[38],相比较而言缺少对面向方面语言基础理论的研究和语言支持的创新。

虽然面向方面编程为运行时验证中监控器插桩提供了对于插桩位置描述和模块化管理的高效机制,但由于面向方面编程本身是从面向对象编程发展而来,目前研究的重点基本都在于对 C++ 和 Java 等面向对象语言的支持。然而在许多嵌入式系统和安全关键系统中,基于对运行效率或内存控制的要求,都是采用 C 语言进行开发的,需要一个系统的 AOP 支持语言来支持 C

语言的运行时验证中监控器插桩。

研究如何通过源代码自动插桩实现监控器和验证器的自动集成，并通过实现面向过程 C 语言的 AOP 支持，完成监控器集成位置描述和模块化管理，对于促进运行时验证实际应用，提高嵌入式系统和安全关键系统的可靠性有特别重要的意义；同时，它能为设计和实现对其他面向过程语言的 AOP 支持提供参考和帮助；提高 C 语言系统的开发效率。

1.3 论文研究内容

本课题结合运行时验证对于监控器和验证器集成的应用需求背景，综合运用编译器源代码插桩方法和 AOP 面向方面编程进行了设计和实现。文章首先通过系统地设计和实现针对 C 程序的 AOP 支持语言 MOVEC，解决对 C 程序运行时验证中监控器集成位置描述，模块化管理和自动插桩集成。同时基于编译器源代码插桩的思想，实现了三种常见软件错误（除 0 错误，整数溢出和变量使用前未初始化）验证器的设计和插桩。

本文的主要研究内容组织如下：

第一章介绍运行时验证中监控器和验证器集成的研究背景及意义，分析当前运行时验证监控器和验证器集成方法的研究现状，陈述选择代码插桩和 AOP 编程方法作为研究主题的选题依据，最后列出本文的主要研究内容和文章结构。

第二章介绍编译相关知识和相关工具，主要针对 C 语言的编译流程进行分析，描述用于代码结构分析的 LLVM 编译器构成，及其抽象语法树（Abstract Syntax Tree）的结构，介绍用于实现自定义语言编译器生成的 Flex 和 Bison 的使用方法。

第三章介绍面向方面的传统关注点分离模型，面向方面编程概念和面向方面编程的技术关键，最后将现有针对 C 程序的 AOP 支持语言进行总结对比。

第四章介绍本文所设计的针对 C 程序 AOP 支持语言 MOVEC 的语法，其相应编译器编译流程，以及具体的实现细节。

第五章介绍在三种常见错误的软件运行时验证中，验证器的设计，自动化代码插桩原理及其具体实现细节。

第六章在第四章设计实现的 MOVEC 编译器的基础上，将其应用于 MiBench 嵌入式基准测试集合，在正确性，时间性能和空间性能方面和现有 AOP 语言进行对比测试，记录实验结果，并对实验数据进行分析 and 总结。同时，通过实验验证常见错误验证逻辑和插桩方法的正确性。

第七章总结本文的研究内容，并对未来进一步的工作进行展望。

第二章 编译知识和相关工具的介绍

2.1 语言的编译流程

编译器是指一种能将源程序映射成为语义相同的目标程序的工具，由分析和合成两个部分组成，其中分析过程将源程序分解为各个部分并通过语法结构描述各部分之间的关系，合成过程利用中间代码和符号表的信息重新构造所需要的目标程序。更具体一点说，通常编译器的编译流程可以划分为如下七个阶段：词法分析，语法分析，语义分析，中间代码生成，中间代码优化，机器代码生成，机器代码优化，如图 2.1 所示^[39]。

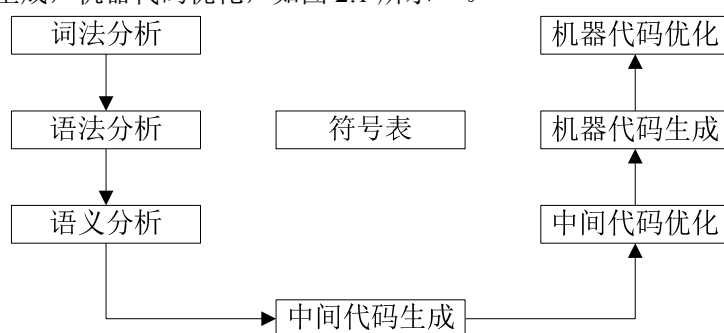


图 2.1 编译器编译流程

在词法分析阶段，编译器读取程序文本，并按照特定的词法规则将其分割为具有特定意义的字符序列，即词法单元，每一个词法单元对应一类标志符号。在语法分析阶段，编译器读取词法单元，按照特定的语法规则将其组织成语法树的形式，通过语法树反映出整个程序的结构。语义分析是在语法树的基础上，利用符号表的信息，对代码中可能存在违反语言定义的情况进行检查，例如函数参数类型不匹配等，操作数类型不匹配，同时该阶段会将类型信息记录在抽象语法树或符号表中，为后续阶段服务。中间代码生成可以理解为在一个抽象的通用机器生成等价的程序，中间代码的设计要求易于生成和转化为实际的机器语言。中间代码优化是利用特定的优化技术，例如编译时转换，基于数据流的优化等，保证目标程序获得更快的运行速率，更低的空间占用或者更低的能源消耗。机器代码生成部分将中间代码映射成为特定语言的程序，特别地，如果目标语言为机器语言，中间代码里的变量都将被寄存器编号或内存位置所替代，然后输出为汇编代码。

2.2 LLVM 编译器框架

LLVM (Low Level Virtual Machine) 是一个包含了一系列模块化，可重用的编译器和工具组件的编译器框架。其最初是作为伊利诺斯大学的一个研究项目进行的，目标是提供一种基于静态单赋值技术，适用于任意编程语言的静态和动态编译方法，静态单赋值是指每个变量只

能被赋值一次，现代编译器的中间代码设计大多会采用该技术。随着 2005 年苹果公司将其集成到苹果的开发工具中，索尼公司将 LLVM 的前端工具 Clang 引入作为 PS4 的 SDK (Software Development Kit) 的一部分，越来越多的诸如谷歌，英特尔等知名公司加入到对 LLVM 的开发当中，这些都极大的促进了 LLVM 编译器框架对于语言标准支持等功能的改进和在现实开发中的应用。

LLVM 编译器框架主要由三个部分组成，一是 LLVM 核心库，其作用是针对 LLVM 中间代码 (LLVM IR) 提供一系列的分析和优化工具，并将最终的中间代码针对特定的目标平台进行机器代码生成。二是 Clang 编译器前端，Clang 编译器前端提供了将 C/C++/Object-C 代码转换为 LLVM 中间代码的工具，在转换过程还可以对代码进行静态分析。三是一系列相关辅助工具，例如工具 `dragonegg` 将 Gcc 的前端解析器和 LLVM 的后端机器代码生成器进行了结合，利用 Gcc 前端实现了对更多语言的编译支持，工具 `libc++` 通过对 C++ 标准库的高效率实现，保证了 LLVM 对 C++ 最新标准的支持。

LLVM 的整体设计思想是通过在程序编译时，链接时，运行时，空闲时提供高层次的程序结构信息，实现对任意程序透明的，全时段的分析和转换，这是其他现有编译器所不能做到的^[40]。LLVM 的体系结构非常适合用于程序代码的转换，包括源代码级别和中间代码级别，源代码级别转换对应 Clang 编译器前端部分，中间代码级别转换对应 LLVM 核心库部分。源代码转换和中间代码转换相比，源代码包含更丰富的类型和代码结构信息，源代码具有跨平台和二次开发的特性，因此，本文下面将着重介绍 LLVM 中的 Clang 前端的整体结构以及抽象语法树的操作方法。

2.3 Clang 前端和抽象语法树

Gcc 作为编译器前端，由于其结构过于复杂，代码调试信息不易理解，在应用领域方面存在诸多限制，因此 LLVM 重新设计了 Clang 前端以适应整个 LLVM 编译器框架的模块化，可重用化特点。Clang 相比 Gcc 具有如下优点，一是实现了对 C 系列语言的支持，兼容 Gcc，具有清晰的调试信息，二是基于类库的代码结构，便于整个编译器代码的重用和扩展，三是编译器实现了静态分析，代码生成，源代码转换，重构等多重功能，四是编译过程有更低的内存消耗和时间消耗。

Clang 编译器主要由 11 个模块组成，`support` 和 `system` 模块定义继承于 LLVM 的相关基础数据类型，`basic` 模块定义调试信息，源代码位置，源代码缓存等辅助类型，`ast` 模块定义抽象语法树表达所需的类型，`lex` 模块定义预处理和词法分析过程，`parse` 模块定义语法分析过程，`sema` 模块定义语义分析过程，`codegen` 模块定义抽象语法树与中间代码的转换关系，最终的代码转换，`rewrite` 模块定义了代码插桩和重构方法，`analysis` 模块实现了静态分析相关支持，`driver`

模块定义了编译器的入口和相关选项处理。

在一个 C 语言项目中，一个源文件和其包含的头文件构成了一个编译单元，Clang 经过词法分析和语法分析阶段，将分析获得的抽象语法树和符号表存储在 `ASTContext` 类对象中，通过 `getTranslationUnitDecl` 方法可以获得整个抽象语法树的根节点。

Clang 的抽象语法树由众多不同的节点类对象构成，这些节点主要可以分为三类，声明类节点，语句类节点和类型类节点。虽然这三类节点都采用了继承的层次结构，但是拥有不同的继承根节点，声明类节点以 `Decl` 类和 `DeclContext` 类作为继承根节点，语句类节点以 `Stmt` 类作为继承根节点，类型类节点以 `Type` 类作为继承根节点。

图 2.2 以 `VarDecl` 类为例说明声明类的继承结构。`VarDecl` 是变量声明类，因为变量声明是可重复声明的部分，所以需要继承 `Redeclarable` 模板进行特殊扩展，另一方面，变量声明是带有声明符信息，所以需要继承 `DeclaratorDecl` 声明符声明。`DeclaratorDecl` 自身存储了声明符号相关信息，通过继承 `ValueDecl` 实现对变量值属性的描述。`ValueDecl` 自身存储了变量的类型信息，通过继承 `NamedDecl` 实现了对变量名称属性的描述。`NamedDecl` 自身存储了变量的名称信息，通过继承 `Decl` 节点表明这是一个对变量的声明。在 `VarDecl` 基础上还可以通过继承得到 `ParmVarDecl` 参数变量声明等。

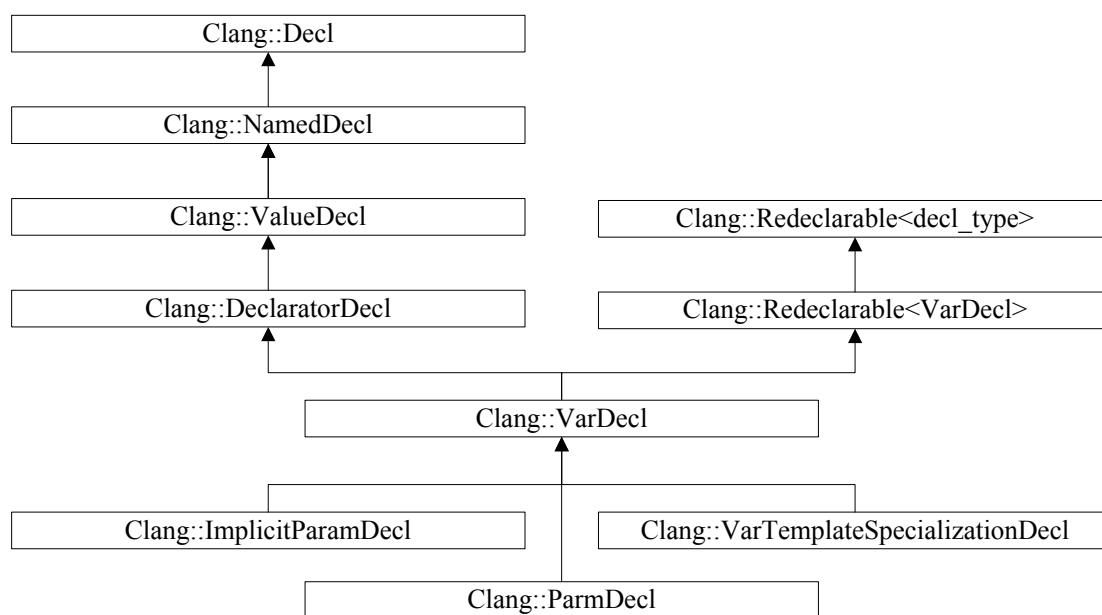


图 2.2 Clang 中声明类继承结构

图 2.3 以 `CompoundAssignOperator` 类为例说明语句类的继承结构，特别需要说明的是，在 Clang 的抽象语法树结构中，表达式被归类为语句的一种特殊形式。`CompoundAssignOperator` 是复合赋值表达式类，自身存储了计算结果类型等信息，因为复合赋值操作符首先是一个二元操作符，所以需要继承 `BinaryOperator` 节点。`BinaryOperator` 二元操作符表达式自身存储了二元操作符类型和两个操作数，通过继承 `Expr` 节点实现了复合赋值属于表达式的属性。`Expr` 在存

除了表达式结果类型的基础上，继承了 Stmt 节点。除了表达式节点，在 Stmt 节点基础上还可以继承得到 DeclStmt 声明语句，ForStmt 循环语句等语句类节点。

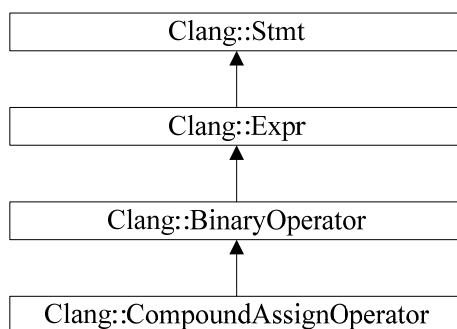


图 2.3 Clang 中语句类继承结构

图 2.4 以 FunctionProtoType 类为例说明类型类的继承结构。FunctionProtoType 是带函数原型信息的函数类型类，在存储了函数原型信息基础上，需要继承 FunctionType 表明它的函数类型属性。FunctionType 存储了函数返回类型等信息，并通过继承 Type 类型，表明这是一个类型节点。

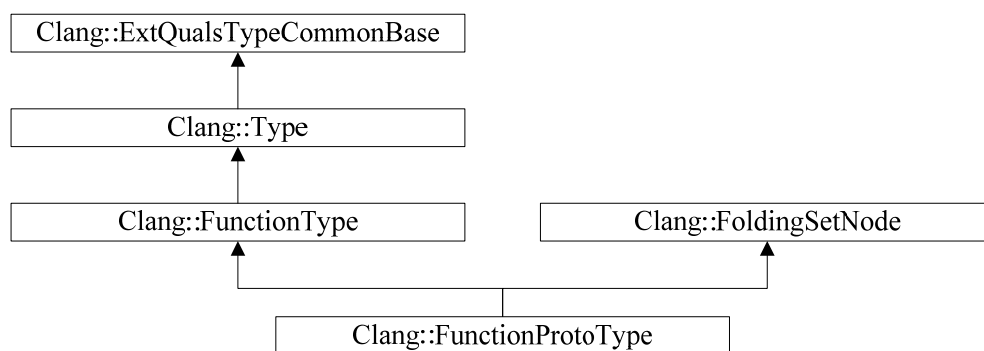


图 2.4 Clang 中类型类继承结构

通过这样的继承结构，使各种不同的节点类型之间有了明显的层级区分，又能够通过类型的复用，每个类型分别存储相关信息，降低整个语法树结构的复杂度。

在获得抽象语法树的基础上，Clang 提供了一个 RecursiveASTVisitor 模板类，可以用于以任意节点为根节点的抽象语法树遍历。RecursiveASTVisitor 类是一个前序深度优先遍历装置，可以通过 Traverse 类函数完成整个抽象语法树节点的遍历，在任意语法树节点，可以通过 WalkupFrom 类函数遍历该节点的类型继承树，在类型继承树上的任一个节点，可以通过 Visit 类函数实现对任意类型节点的访问操作，这三类函数可以通过继承 RecursiveASTVisitor 类并重写以实现自定义访问。

Clang 中的 rewriter 模块提供了代码插桩和重构的方法。Rewriter 类提供了对任意源代码位置执行插入，删除和替换的能力，其中的源代码位置来自于 SourceManger 对于源代码的定位管理，在实际写回到文件之前，所有对源代码的改动内容和改动位置都将被记录在 RewriteBuffer

类对象中。

在基于 Clang 的编程接口方面, Clang 提供了三种类型的接口。LibClang 是一个稳定的 C 语言接口, Clang Plugins 作为一种插件随 Clang 编译器的运行而运行, LibTooling 是一个 C++ 语言接口, 可以用于编写独立运行的分析转换工具, 而且其对抽象语法树的访问能力优于 LibClang。LibTooling 的原理为针对输入的源文件和编译参数, 按照既有的 Clang 编译器进行预处理, 词法分析, 语法分析, 这些过程对应 FrontendAction 类的 ExecuteAction 函数执行, 开发者可以通过重定义 FrontendAction 为 AST 树的生成过程配置回调函数类, 即 ASTConsumer, 在 AST 树生成完成后利用 RecursiveASTVisitor 进行遍历和自定义访问, 如图 2.5 所示。

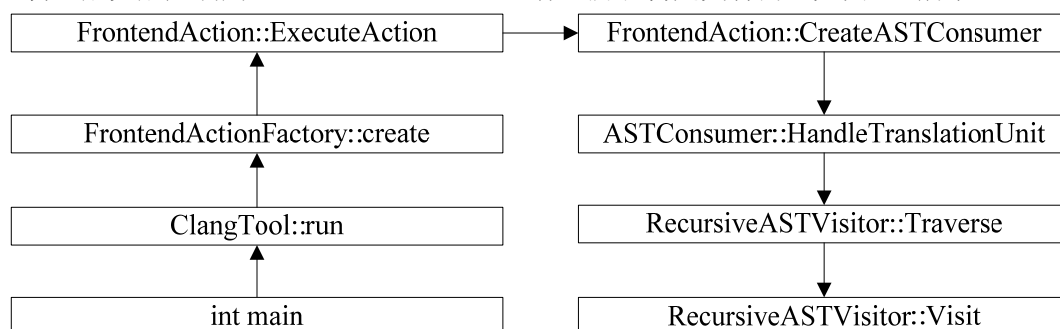


图 2.5 ClangTool 执行流程

2.4 Flex 词法分析生成器和 Bison 语法分析生成器

Flex 和 bison 是用于生成分析程序的工具, 非常适用于处理结构化的输入, 在设计自定义编译器等众多领域有着广泛的应用^[41]。最初使用的语法分析生成器是由贝尔实验室开发的 yacc, 1985 年伯克利大学的 Corbett 改进了其内部算法, 重新对 yacc 进行了实现, 后来自由软件基金会对 yacc 进行了再次改写并将其纳入了 GNU 项目进行维护, 最终 yacc 在不断的改进之后成为了现在 Bison。Lex 的最初版本由 Lesk 和 Schmidt 在 1975 年编写完成, 1987 年伯克利实验室的 Paxson 重新利用 C 语言对 lex 进行了重写, 即 flex, 相比原有的 lex 更加快速和可靠, 目前 flex 作为 SourceForge 项目进行开发和维护。

Flex 的输入文件包含三个部分, 每个部分之间用单独的一行%%进行分隔。第一部分是定义部分, 用于定义一些特定的变量或导入头文件, 这部分的代码会被直接输出到词法分析器的代码中。第二部分是规则声明部分, 包含了一系列的模式动作映射, 模式采用扩展的正则表达式进行描述, 动作是对应模式得到匹配所要执行的代码。第三部分是用户代码部分, 用于定义调用词法分析功能或被词法分析过程调用的函数, 这部分的代码也将被直接输出到词法分析器的代码中。一个简单的用于统计字符数和行数的 Flex 输入文件如图 2.6 所示, 其中 yylex 函数为对词法分析过程的调用。

用户利用 Flex 对输入文件进行处理之后, 会得到一个 C 程序文件, 再利用 C 语言编译器

进行编译，便可以得到一个高效率的词法匹配分析工具，该工具的输入可以是标准输入，也可以读取文件作为输入。

```
int line_count = 0, char_count = 0;
%%
\n      ++ line_count;  ++ char_count;
.       ++ char_count;
%%
int main() {
    yylex();
    printf("# of lines = %d, # chars = %d\n", line_count, char_count);
}
```

图 2.6flex 输入文件示例

Bison 的输入文件主要由四个部分组成，第一部分由%{和%}符号包含，剩余的三个部分由单独的一行%%进行分隔。第一部分是通用声明部分，包含对语法规则描述所要用到宏的定义和变量函数的声明。这些代码将被直接输出到生成的语法分析器代码的开头位置。第二部分是bison 声明部分，用于声明语法规则中所用到的终结符和非终结符，指定优先级等。第三部分是语法规则说明部分，包含用巴科斯范式形式编写的一系列语法规则，每条语法规则中的语法符号都有相对应的语义值，可以通过附加动作利用语义值实现特定的功能。第四部分同样是用户自定义代码部分，这部分的代码将被直接输出到生成的语法分析器代码的结尾位置，在这里可以调用语法分析过程或定义其他函数，一个浮点数加法计算器语法文件如图 2.7 所示。

```
%{
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER ADD SUB EOL
calclist:
| calclist exp EOL { printf("= %f\n", $2);};
exp: factor
| exp ADD factor { $$ = $1 + $3;};
%%
main(int argc, char **argv) {yyparse();}
```

图 2.7bison 输入文件示例

用户利用 Bison 对输入文件进行处理之后，会得到一个 C 程序文件和头文件，将其和相应

的词法分析器文件进行联合编译，便可以得到一个简单的加法计算器，能输出加法计算结果。

2.5 本章小结

本章首先介绍了编译的相关概念，包括整个编译过程的组成和每个编译阶段的输入输出，因为本文中涉及的面向方面语言的设计和基于编译器的代码插桩都是会利用到程序编译的概念。

其次描述了开源编译器框架 LLVM 的整体结构，着重介绍了其中的前端工具 Clang，抽象语法树在 Clang 中的存储，定义，访问和修改方法。LLVM 和 Clang 的模块化结构，基于程序分析和变换的设计思想，非常适用于代码插桩的实现。

最后介绍了 Flex 词法分析工具生成器和 Bison 语法分析工具生成器的使用方法，通过定义词法规则和语法规则便可以实现一个特定的编译器，适用于本文后续针对 C 程序 AOP 支持语言的实现。

第三章 面向方面语言

3.1 传统关注点分离模型

正如绪论 1.2 节中所述,面向方面编程的提出,是在面向过程编程和面向对象编程的基础上发展而来的,关注点分离的概念最早由 Tarr 和 Ossher 于 1999 年提出^[42],面向过程编程和面向对象编程一直在解决软件中垂直结构关系中的关注点分离问题,即核心关注点,面向方面思想提出的目标是解决软件中横切结构关系中的关注点分离,这些横切关注点使系统存在代码纠缠(一个模块实现了多个关注点)和代码散乱(一个关注点由多个模块实现)的问题。为了解决这一问题,研究人员提出了一系列的模型,作为横切关注点的实现方法,下面将对这些传统模型进行简单的介绍。

3.1.1 自适应面向对象编程

在传统的面向对象编程中,一个类的定义是通过类结构和成员方法描述而实现的,自适应面向对象编程继承了这样的定义方法,所不同的是,自适应面向对象编程对类结构的定义属于部分定义,只是给出了类结构所需要满足的限制条件,成员方法也未直接实现,只有在真正调用该成员方法的时候才会被指定实现内容^[43]。

自适应程序由一系列的传播模式组成,通常一个传播模式包含一个操作分句,一个横穿分句和一个代码分句集合。操作分句定义了该传播模式所要表达的方法名,横穿分句通过路径限制的方式定义了包含该模式方法的类,代码分句定义了该模式方法的具体实现。定义了基本功能和横切功能的传播模块共同组成了一个完整的自适应程序。

对于一个模式编译器,输入一系列的模式和一个类结构图,就会输出一个面向对象的程序。具体包括如下三步:1,搜索获得模式中涉及的类名,称为模式相关类,从类结构图中计算得到一个包含所有模式相关类的子图;2,为所有的模式相关类增加一个操作分句所定义的函数;3,将代码分句中的实现填充到新增加函数的代码中。

在自适应面向对象编程中,每一个传播模式都是相互独立的,通过类结构形成一个松散的关联关系。基于这种设计,一个传播模式内代码分句的改变,并不会影响到其他传播模式,从软件的较高层次实现了关注点的分离。

3.1.2 面向主题编程

主题是指从某个特定的角度对系统状态和行为规范描述的集合。例如,对于同样一个杯子,普通使用者可能关注的是杯子的容积和喝水的功能,而一个艺术家可能关注的是杯子的颜色,

形状和装饰的功能。对于一个系统，可以由众多的主题进行描述，每个主题包含了许多类，这些类代表系统的某些组件，类中定义了从该主题角度组件具有的状态和功能，这些主题的所有状态和功能共同构成了一个完整的系统^[44]。

面向主题编程是建立在面向对象编程基础之上的编程方法，因此面向主题编程继承了面向对象编程中的某些概念，但也有所不同。例如接口是一类对象所支持的操作的抽象化描述，通过实现方法描述和方法实现的分离，能够使对象所支持的方法在某一个主题中实现即可，其他的主题只需要进行声明，同时这些主题也不需要知道具体哪个主题中进行了实现。行为是指一类对象所支持的操作的定义描述，传统面向对象编程中，方法实现是与类绑定的，但是在面向主题编程中，对于同一个类，不同的主题中可能支持不同的操作。与之相类似的，对于同一个类，不同的主题中可能包含不同的用于存储状态信息的实例化变量。

面向主题编程的目标是解决类的接口，方法定义和状态的融合，因此需要定义一系列的组合规则以实现主题间的融合。组合规则主要分为如下三类：一是方法调用组合，因为一个主题中的方法调用可能导致其他主题中的方法执行；二是对象创建和初始化组合，因为一个主题中的对象创建初始化可能导致其他主题中的相关初始化；三是状态引用组合，因为不同的主题之间可能存在数据共享，一个主题中状态变量的修改可能也会影响到其他主题的状态变量。对于不同主题之间对象匹配问题，在对象创建和初始化的时候，会生成一个对象标识符 oid，不同的主题中相同的类会拥有相同的 oid，这样在处理方法调用组合的时候便可以通过搜索 oid 完成跨主题的方法调用。

面向主题编程通过从不同角度描述系统的方法实现了关注点的分离，又通过组合规则实现了关注点实现的融合，是一种有效的关注点分离模型。

3.1.3 元对象编程

在元对象编程中，基本级对象定义了程序的基本算法，元级对象可以对程序的基本功能进行修改，例如元对象可以截留发送给基本级对象的消息，并加以同步性质的判断，或进行相应的输出，基本级对象和元级对象之间的联系通过元对象协议实现^[45]。

具体到元对象编程的实现方法，其可以采用反射的机制。整个系统由两层结构组成，在底层结构中，定义了系统中的类结构和对象调用，而每一个底层对象都会通过反射机制绑定一个元级对象，元级对象负责定义底层结构中的操作。

元对象编程利用反射，使元级对象的功能可以实现运行时动态自定义，具有了更高的灵活性，而基本级对象也可以关注于自身的主体逻辑实现，实现了关注点的分离，提高了系统的扩展性。

3.1.4 组合过滤编程

组合过滤编程是对传统面向对象编程的扩展，在面向对象编程中，一个对象作为类的一个实例而存在，其中包含了基本类型实例，内部类对象实例，外部类对象引用，内部方法等。组合过滤编程在对象中增加了一个或多个过滤器，并通过过滤器实现关注点的分离^[46]。

在面向对象的系统中，由于封装技术的采用，对象仅通过接口对外提供功能调用，而隐藏了具体功能实现，而对象之间的接口调用体现为对象消息的发送和接收。过滤器作为过滤器类的一个实例，分为发送过滤器和接收过滤器，分别作用在消息的发送和接收过程中，决定消息通过的允许和拒绝。

过滤器本身是一个独立于基本功能之外的自定义结构，对于不同的过滤器类型，其在对消息采取允许或拒绝时，可以执行不同的动作。从功能性方面分析，过滤器可以基于对消息类型的分析，将消息进行再调度，发送给特定内部处理函数或内部类对象，或直接拒绝消息的传送，作错误处理。从时序性方面分析，过滤器可以重新定义消息发送或接收顺序，例如在需要同步的系统中，可以根据情况暂时挂起某个消息，在需要实时反应的系统中，可以通过消息中包含的任务消耗时间，截止时间等信息计算其权重，实现消息处理调度等。

程序中的每一个横切关注点都可以对应一个过滤器类，过滤器类实现横切关注点中的功能，并将过滤器对象加入到横切关注点相关的基本功能对象中，组合过滤编程利用过滤器的方式从方法调用和方法执行两个角度完成了横切关注点和核心关注点的分离。

3.2 面向方面编程的基本概念

面向方面编程是由 Gregor 等人于 1997 年在欧洲面向对象编程大会上提出的横切关注点分离方法^[28]，和上述的 4 种模型机制复杂，缺少相应的开发工具支持不同，面向方面编程目前已经实现了对多种编程语言的支持，包括常用的 C/C++，Java，以及一些功能性语言 Matlab 等。其中以基于 Java 的面向方面编程研究最为成熟，并广泛商用于互联网应用开发中。

面向方面编程中的程序包含两个部分，基础代码用于实现系统的核心关注点，方面代码用于实现横切关注点，经过相应方面语言的编译器编译之后，输出组合了核心关注点和横切关注点的目标程序代码。面向方面编程中采用的是接入点模型，模型由接入点，切入点，通知和方面等组成，下面将分别对这些结构进行介绍。

接入点是指在程序结构中的特定位置或运行过程中的特定时刻，在这些位置可以织入横切关注点的功能。通常意义上，连接点被分为静态连接点和动态连接点两类，静态连接点是指程序静态结构中的位置，如函数和类的声明等，而动态连接点是指程序动态运行过程中特定动作的时间点，如函数的调用，变量的赋值等。

切入点是指符合一定条件的连接点的集合，通过切点表达式进行描述。和连接点对应的，

基本切入点也可被分为基本静态切入点和基本动态切入点，在基本切入点的基础上，往往可以通过运算符，构造出复杂条件的切入点。切点表达式通常由一系列配套的语法元素组成，并按照特定的语法规则进行多种组合，共同构成切点表达式。切点表达式的使用既规范了对接入点的描述，又提高了接入点使用的灵活性。

通知是指在切入点所描述的特殊接入点位置，需要执行的一系列动作，即横切关注点的目标执行动作。通知和接入点之间的关联关系通常有如下三种：**before**，**around** 和 **after**，分别表示在进入接入点之前执行通知动作，跳过原接入点执行通知动作，在离开接入点之后执行通知动作。在通知中，还可以提供相应的辅助机制，重新进入跳过的接入点，通过特殊结构获取接入点的上下文信息，例如当前调用函数的名称，返回类型，参数类型，实际参数值和实际返回值等。

切入点和通知共同构成了一个关注点的模块，而若干个关注点的集合则构成了一个方面。通常，方面具有和类相似的结构，切入点类似于面向对象中的变量声明，通知类似于面向对象中的方法声明，方面的结构设计使对横切关注点的管理更加系统，同时更容易进行扩展。

3.3 面向方面编程的技术关键

在面向方面编程中，由于整个接入点模型包括接入点，切入点，通知，方面四个部分组成，因此，面向方面编程设计的技术关键也是围绕这几个部分进行的，下面将针对各个组成部分介绍设计需要解决的问题和现有的解决方法。

3.3.1 接入点设计

在接入点的相关设计中，主要涉及对面向方面编程中所支持的接入点的选择。而接入点的选择主要受两个方面因素的影响，一是基础语言的限制，基础语言的语法结构种类是整个接入点最大可选集合，不同的语言拥有不同语法结构，例如面向过程的 C 语言包含了函数定义，函数调用，函数执行，变量赋值，变量取值等，而面向对象的 C++ 语言则在此基础上增加了类型定义，类型初始化，类型析构等，这样在接入点设计方面，C++ 比 C 语言具有更多的可能性。二是方面语言的应用要求，SpringAOP^[32]和 AspectJ^[29]都是针对 Java 语言的面向方面语言扩展，但是由于二者的应用场景不同，SpringAOP 在企业级应用中配合 Spring 的 IOC 控制反转功能，所以其只支持方法执行的接入点，AspectJ 的设计目的是作为一个通用的面向方面语言，所以其在接入点的设计上涵盖了函数执行，函数调用，属性取值和属性赋值等各个方面，另外 AspectMatlab^[36]为了实现对 Matlab 科学计算编程具有重要意义的循环支持，增加了 for 循环接入点。面向方面语言中接入点的选择和设计，直接决定了整个面向方面语言对横切关注点的支持程度及其扩展性。

3.3.2 方面表达设计

在切入点，通知和方面的表达设计中，主要涉及对切入点，通知和方面的语法结构选择，及方面代码与源代码的相互关系选择。根据语法结构，方面表达语言可以分为两类，即非对称性语言和对称性语言；根据方面与源代码的关系，方面表达语言又可以分为两类，即嵌入式语言和独立式语言^[47]。

非对称性语言是指方面语言和源代码采用不同的语法结构，因为面向方面编程的概念是发展自面向对象编程，设计目的也是为了解决面向对象编程中所未能实现的横切关注点的模块化描述，所以大多数面向方面语言的语法结构也是基于面向对象语言的扩展，通过增加相应的关键字和语法规则，实现对切入点，通知和方面的描述。例如 AspectJ^[29]，就是在 Java 语言语法的基础上增加了 aspect, pointcut, calls, executions 等关键字及相关语法结构，aspect 表示方面，pointcut 表示切入点，calls 和 executions 分别表示函数调用和函数执行。

对称性语言是指方面语言和源代码采用相同的语法结构，利用类，成员函数，成员变量等既有语法结构完成切入点，通知和方面的描述。例如在 InterAspect^[48]中，切入点成为一种特殊类型的结构体，并通过相应的接口函数生成，函数调用切入点就是通过 aop_match_function_call 函数生成，而切入点相关参数的指定，同样通过接口函数实现，aop_filter_call_pc_by_name 可以用于指定函数调用切入点中的函数名称，aop_filter_call_pc_by_return_type 可以用于指定函数调用切入点中的返回类型。

嵌入式语言是指方面的相关描述直接嵌入于源代码文件中，通过人为在特定代码位置增加注释等方法，完成对切入点，通知和方面的描述。Annotation（注解）是 Java5 版本中引入的新特征，可以对源代码中的任意类，方法，变量或者参数进行注解，和 Javadoc 不同的地方在于注解运用了反射机制，注解中的内容可以被编译进 class 文件，系统运行时通过反射访问注解中的元数据，Java 的注解主要用于文档生成，代码静态分析和编译检查。针对 Java 的 AOP 语言 Aspectwerkz^[30]正是利用了 Java 的注解机制，当采用如 @Around("execution(* com.aspect.testcase01.target.*(..))") 形式语句，对一个成员函数如 log(ProceedingJoinPoint pjp) 进行注解，便可以实现对 target 类中所有函数执行的横切，跳过原有执行流程，转而执行 log 函数。

独立式语言是指方面的相关描述独立于源代码，存储在特定的文件中，这种设计最大可能实现方面代码的重用，同时增加了方面代码使用的灵活性。在 Spring AOP^[32]中，其对方面的描述综合运用了 Java 和 xml 语言，首先在 Java 文件中利用函数完成通知目标动作的定义，与普通函数不同的是，函数包含类型为 Joinpoint 的参数，然后在 xml 文件中完成切入点定义以及切入点和通知函数映射关系的描述。例如 <aop:pointcut id="logpointcut" expression="execution(* com.service.UserService.*(..))" /> 定义了一个 UserService 类中任意函数执行的切入点，配合 <aop:before method="beforeAdviceOne" pointcut-ref="logpointcut" /> 的映射定义，便可

以实现在 UserService 类中任意函数执行前调用 beforeAdviceOne 函数。

3.3.3 切入点匹配设计

切入点是对符合特定条件的一类接入点的抽象化描述，接入点包括代码静态结构中的特定位置或程序执行流程中的特定时刻，对于面向方面编程来说，需要在特定接入点执行特定的通知目标代码，所以需要实现方面代码中的切入点和程序中接入点的匹配，这是面向方面编程正确性的基础。

根据切入点匹配的时间可以分为编译时切入点匹配和运行时切入点匹配。

编译时切入点匹配是指在编译过程中，通过对抽象语法树，中间代码等不同的程序结构进行分析，将切入点匹配到这些程序结构中的特定位置。例如 AspectJ^[29]便是通过对编译生成的字节码进行分析，访问所有可能的接入点位置，并将切入点描述和接入点的性质进行匹配，匹配成功的接入点便称为切入点在程序中的投影，后续会在这些位置执行对通知目标动作的调用。能够进行程序结构分析的对象除了编译过程中生成的抽象语法树，中间代码之外，还可以是一些自定义的数据结构，例如 AspectC++^[34]中引入了知识库的概念，在利用编译器编译源文件生成抽象语法树的基础上，根据其所支持的切入点种类，找出所有可能的接入点，在知识库中记录接入点的类型，名称，参数，所在文件位置等相关信息，这样在后续的匹配过程中，便可以直接将切入点和知识库进行匹配，这种匹配方式一方面通过知识库避免同一个文件多次匹配带来的重复编译，提高了重用性，另一方面使程序中的接入点能够可视化，方便用户的调试。

运行时切入点匹配是指在程序执行过程中，通过对程序运行过程中的相关状态信息进行分析，继而将切入点匹配到程序运行过程中的特定时间点，这种方式相比较于编译时切入点匹配会对程序执行效率产生消耗，但使得切入点的使用更加灵活，产生事件的程序代码具有重用性。例如在 EAOP^[49]中，其引入了事件的概念，事件是指程序的特定执行点，目前 EAOP 中支持的事件包括函数的调用及其返回事件，构造函数调用及其返回事件。预处理器会在源程序中插入事件生成语句，由执行监控器在代码运行过程中对事件进行接收并传递给其所管理的方面实体，方面实体作为一个转换器，针对事件中内容和方面中切入点进行匹配，并在此基础上完成对相应通知代码的调用。

3.3.4 通知调用织入设计

在切入点匹配的基础上，需要执行切入点相应的通知目标动作，因此在面向方面编程中，通知调用织入的设计是保证面向方面编程正确性的重要组成部分。

根据通知调用织入实际发生时间，通知调用织入可以分为通知调用编译时织入，通知调用运行时织入。

编译时通知调用织入是指在编译过程中，利用编译器既有的代码改写支持或其他的代码转

换工具，将对通知的调用直接插入到程序的特定结构位置，例如 AspectJ^[29]是将通知调用以字节码的形式插入到 class 文件中，AspectC++^[34]是将通知调用以源代码的形式插入到 C++源文件中。因为字节码或源代码插桩的形式与环境无关，织入的结果可以实现跨平台的使用，另外通过编译时指定通知调用，降低了通知调用选择对程序执行效率的影响。

运行时通知调用织入是指，在程序实际运行之前，程序中并不包含对通知代码的调用，而是在程序运行时完成切入点的匹配，通知的选择和调用，例如 SpringAOP^[32]利用了 Java 的反射机制，当程序运行到特定的连接点，获取其类型和上下文信息，在此基础上完成切入匹配和通知调用，EAOP^[49]则是利用事件的形式，向面向方面主程序提供接入点的相关信息，并在此基础上完成切入匹配和通知调用。运行时通知调用织入方式将通知的选择和调用放在程序运行时进行，会部分影响程序原有的运行效率，依赖特定的运行环境，但同时运行时通知调用织入又给予了通知与切入点之间动态绑定的特性，绑定关系在程序运行过程中可以根据需求进行修改。

3.4 针对 C 的面向方面解决方案对比

和 Java 语言相比，在支持 C 的 AOP 语言方面的研究开发依然处于一个较低的成熟度，下面将分别对现有的针对 C 语言的面向方面解决方案进行对比说明。

3.4.1 AspectC

ACC (AspeCt-oriented C) 是目前针对 C 语言的最完整 AOP 支持工具^[33]。对于输入的预处理之后的代码文件，利用工具中的 C 编译器进行解析，同时查询输入的方面文件，在匹配的代码位置插入相应的横切关注点代码。ACC 的不足之处在于：1) ACC 输入的是预处理之后的文件，使得源程序中的文件包含指令在预处理过程中被替换掉，ACC 失去了相关的连接点；2) 程序文件经过预处理之后，不再保持原有的文件结构，所有的头文件被整合到主文件中，带来大量代码冗余，而且不利于用户对后续代码的二次阅读和开发；3) ACC 中采用的 C 语言编译器是自行利用 GENTLE 编译器构造系统开发实现的，在针对特殊语法结构，Gcc 或 Clang 的特殊扩展等方面，往往不能正确识别，另外，该项目的相继停止更新和网站下线，也使得 ACC 编程工具的使用变得更加困难。

3.4.2 AspectC++

AC++ (Aspect-oriented C++) 是目前针对 C++语言的最完整 AOP 支持工具，因为 C++对 C 的兼容，在部分情况下，可以使用 AC++实现对 C 语言的 AOP 支持^[34]。对于输入的源文件，AC++首先利用 puma 编译器进行预处理，分析形成代码连接点知识库，然后通过将知识库中的连接点和方面文件中的切入点进行匹配，在匹配的代码位置插入相应的横切关注点代码。AC++的不足之处在于：1) AC++虽然输入的是源文件，并自动实现了预处理，但 AC++的切入点类

型中并没有考虑到文件包含相关的连接点；2) 在 AC++ 的预处理中，和 ACC 类似的，所有头文件被整合到主文件中，带来大量代码重复，且破坏了项目的整体文件结构，不利于二次开发；3) 在 AC++ 的横切关注点织入实现技术中，大量使用了 C++ 独有的语法结构，例如模板，类等，这使得 AC++ 插桩之后的代码对于 C 语言编译器或者嵌入式的编译器，是无法编译运行的；4) AC++ 中使用的 C/C++ 语言编译器是专门为 AC++ 开发的 Puma 编译器，整个项目在对新语言标准和 Gcc 特殊扩展方面的支持更新较慢。

3.4.3 InterAspect

InterAspect^[48] 是一种以 Gcc 插件形式提供服务的 AOP 工具，用户通过 InterAspect 提供的接口函数来指定切入点，InterAspect 首先利用 Gcc 将程序解析为中间代码 GIMPLE，然后进行连接点和切入点的匹配，横切关注点的织入，最后按照原有流程输出目标代码或二进制代码。InterAspect 的不足之处在于：1) InterAspect 在切入点的描述形式上过于复杂，且不支持函数声明，文件包含等静态切入点；2) GIMPLE 作为中间代码会丢失大量结构信息，限制了可识别连接点的种类；3) InterAspect 的通知设计方面，缺少对 around 类型通知的支持，且通知代码无法获取连接点上下文信息；4) InterAspect 的输出形式为目标代码或二进制代码，不利于程序的二次阅读和开发。

其他针对 C/C++ 语言的 AOP 工具，例如 XWeaver^[50]，WeaveC^[51] 等，由于和上述工具相比，在功能定义上有明显不足，这里不再一一描述。综合上述，现有的针对 C 的 AOP 语言在功能定义和实现方式方面均存在缺失和不足。

3.5 本章小结

由于运行时验证中监控器插桩的需求与软件工程中横切关注点的管理需求存在很多的相似，因此，横切关注点的相关研究能够有助于解决运行时验证中的监控器插桩问题。

本章首先介绍了自适应面向对象编程，面向主题编程，元对象编程，组合过滤编程这四种传统横切关注点分离模型，这些模型为面向方面编程的提出奠定了基础。

在简述这些传统模型的不足后，介绍了目前应用广泛的面向方面编程基本概念，并通过分类介绍和工具示例，总结了目前实现面向方面编程支持所需求解决的技术关键，作为后续 MOVEC 语言的解决方案设计的基础。

本章最后将现有的针对 C 程序的面向方面支持工具进行介绍，分析其不足之处，为针对 C 程序的面向方面 MOVEC 语言的语法设计和实现，提供对比和参照。

第四章 MOVEC 语言设计与实现

4.1 MOVEC 语言的语法设计

下面将分切入点，通知，方面三部分介绍本文所设计面向方面语言 MOVEC 的语法。

4.1.1 切入点

切入点通过关键字 `pointcut` 进行声明，切入点分为基本切入点和复合切入点，而基本切入点包括静态基本切入点和动态基本切入点两种类型，静态基本切入点对应程序静态结构中的位置描述，动态基本切入点对应程序动态运行过程中特定动作时间点描述。

在静态基本切入点和动态基本切入点中都使用到了代码特定程序结构的匹配，MOVEC 中主要是采用字符串或者带通配符的正则表达式进行匹配模式描述。通过字符串匹配变量名称，类型名称，函数名称等，表达式则是在字符串的基础上，通过通配符的使用，进一步提高模式的表达和匹配能力，`%`可以匹配任意长度的字符串，`...`可以匹配任意个数的参数。例如，通过文字 `int calc(int x)`可以完全匹配一个返回类型为 `int`，函数名为 `calc`，只有一个类型为 `int` 名称为 `x` 的参数的函数；通过引入通配符，正则表达式 `% calc(int x, ...)`，可以完全匹配一个返回类型为任意类型，函数名为 `calc`，第一个参数类型为 `int`，第一个参数名为 `x`，剩余参数的个数，类型，名称均无限制的函数。

在静态基本切入点的定义中，主要是通过结合使用预定义的关键字和特定程序结构描述，完成对切入点静态位置或范围限制的表达。MOVEC 中的静态切入点关键字及其意义如表 4.1 所示。

表 4.1 静态切点关键字

关键字	结构	意义
<code>include</code>	<code>include(identifier)</code>	文件引入
<code>define</code>	<code>define(identifier)</code>	宏定义
<code>expand</code>	<code>expand(identifier)</code>	宏展开
<code>infunc</code>	<code>infunc(function-signature)</code>	在函数定义体内
<code>intype</code>	<code>intype(identifier)</code>	在类型定义体内
<code>infile</code>	<code>infile(identifier)</code>	在文件内

在动态基本切入点的定义中，同样是通过结合使用预定义的关键字和特定程序结构描述，完成对切入点动态执行动作或动作范围的表达，MOVEC 中的动态切入点关键字及其意义如表

4.2 所示。

表 4.2 动态切入点关键字

关键字	结构	意义
动作类型		
call	call(function-signature)	函数的调用
callp	callp(function-signature)	函数指针的调用
execution	execution(function-signature)	函数的执行
set	set(variable-signature)	变量赋值
get	get(variable-signature)	变量取值
动作范围		
inexec	inexec(function-signature)	函数执行流中
condition	condition(boolean-expression)	满足布尔条件
动作辅助		
returning	returning(identifier)	返回值变量命名

其中，在 call 切入点的描述中，除了可以使用全字符串进行完全匹配和通配符进行扩展匹配外，还支持对函数参数列表的简略表达，即不直接指定参数名称。例如 `call(double func(int, int))`，表示对一个返回类型为 `double`，名称为 `func`，参数为两个 `int` 类型变量的函数的调用，而参数名称为任意。

在 callp 切入点的描述中，由于函数指针的匹配是基于函数名的，所以其中的函数名称必须使用字符串明确指定而不能使用通配符。例如 `callp(double func(int x))`，表示对实际指向返回类型为 `double`，名称为 `func`，参数类型为 `int`，参数名称为 `x` 的函数指针的调用。

在 execution 切入点和 inexec 切入点的描述中，由于库函数的源代码为不可修改的，所以其中的函数不可以指定为库函数等系统函数。例如 `execution(double func(int x))`，表示一个返回类型为 `double`，名称为 `func`，参数类型为 `int`，参数名称为 `x` 的函数体的执行。

在基本切入点的基础上，通过引入逻辑操作符，可以进一步提高切点表达式的描述能力。 $P1 \ \&\& \ P2$ 表示满足 $P1$ 切入点性质连接点和满足 $P2$ 切入点性质连接点的交集， $P1 \ || \ P2$ 表示满足 $P1$ 切入点性质连接点和满足 $P2$ 切入点性质连接点的合集， $!P1$ 表示满足 $P1$ 切入点性质连接点的补集， $(P1)$ 通过引入括号可以实现切入点的更准确的分组和描述。

在 MOVEC 中支持对切入点函数的定义，切入点函数可以包含参数，参数来自于 call 类 pointcut 中函数的参数或者 set 类 pointcut 中的变量名。例如 `pointcut pc1(a) = call(int func(int x:a)) &\& infunc(int main())`，定义了一个名称为 `pc1` 的切入点函数，函数包含一个参数 `a`，对应于函数 `func` 参数 `x` 的重命名。

4.1.2 通知

通知通过 `action` , `advice` 或 `event` 关键字进行声明, 主要由切入点的引用, 目标动作的定义和二者之间的关联关系定义三部分构成, 通知定义的语法如图 4.1 所示, 其中 `[]` 表示可选项, `<>` 表示复杂结构的引入。

```
("action" | "advice" | "event") [actionid "(" <param-list> ")" ] ]
("before" | "after" | "around" ) ["extend"] <pointcut>
("{ " <action-code> "}" | ";" )
```

图 4.1 通知语法结构

由于切入点的定义分为了静态切入点和动态切入点两种, 因此通知也分为静态通知和动态通知两种类型。

静态通知中定义的目标动作作为编译时目标动作, 主要是指利用 `extend`, `before` 等关键字对函数结构, 文件结构或者结构体类型等进行扩展, 或在文件包含指令, 宏定义位置进行代码插桩, 对于编译时目标动作和切入点的关联关系定义包括 `before`, `around`, `after` 三种。例如 `action extend before infunc(% func(...)) {int x;}` 表示将目标动作在 `func` 函数定义的首部进行扩充, `action after define(MACRO_ONE) {two}` 表示将目标动作在宏 `MACRO_ONE` 定义的尾部进行扩充。

动态通知中定义的目标动作作为运行时目标动作, 主要是指在程序运行过程中对程序的运行流程进行修改, 对于运行时的目标动作和切入点的关联关系定义包括 `before`, `after` 和 `around` 三种, 分别表示在特定的时间点之前执行目标动作, 在特定的时间点之后执行目标动作, 替换特定时间点正在执行的动作转而执行目标动作。例如 `action before call(% func(...)) {}` 表示在 `func` 函数调用之前执行目标动作, `action around call(% func(...)) {}` 表示跳过 `func` 函数的调用并转而执行目标动作。

在通知的定义中, 可以包含参数列表, 其中的参数来自于 `call` 切入点中的参数, `execution` 切入点中的参数或 `set` 类切入点的变量, 这些参数可以两种形式为通知的目标代码所用。一种形式为值传递的形式, 例如图 4.2 中 `act1` 通知的定义, `func` 函数的参数以值的形式成为通知的参数, 通知的目标代码中可以进一步使用参数 `x` 的值。另一种形式为指针传递的形式, 例如图

```
action act1(x) before call(int func(int x)) {
    printf("before call func, x=%d\n", x);
}
action act2(*x) before call(int func(int x)) {
    printf("before call func, x=%d\n", *x);
}
```

图 4.2 参数化通知

4.2 中 `act2` 通知的定义, `func` 函数的参数以地址的形式成为通知的参数, 通知的目标代码中可

以进一步使用 `func` 参数 `x` 的值或对 `func` 参数 `x` 的值进行改写。

在目标动作代码中，可以通过预定义的 `struct join_point *tjp` 指针访问当前接入点的上下文信息，具体的上下文信息类型如表 4.3 所示，其中 `args` 参数是指切入点表达式所匹配到的函数参数或切入点表达式所匹配到的变量。

表 4.3 接入点上下文信息

上下文信息类型	意义
<code>type</code>	当前连接点的种类
<code>target</code>	目标函数名或变量名
<code>target_type</code>	目标函数返回类型或变量类型
<code>args[i]</code>	第 <code>i</code> 个参数的地址
<code>args_type[i]</code>	第 <code>i</code> 个参数的类型
<code>args_cnt</code>	参数的总数
<code>ret</code>	返回值的地址
<code>file_name</code>	当前连接点所在文件名
<code>func_name</code>	当前连接点所在函数名
<code>loc</code>	当前连接点所在行号

最后，由于在 `around` 类型的通知中，程序将跳过当前连接点动作，转而执行通知中所定义的目标动作，所以 `tjp` 指针提供了 `proceed` 函数，这样在目标动作代码中可以重新执行当前连接点动作。

4.1.3 方面

通过 `monitor` 关键字实现对方面的声明，方面中包含了相关变量的声明，切入点的声明和通知的声明，是对所有横切关注点的功能性模块划分。一个用于在每次 `func` 函数调用之后，打印 `func` 函数原有参数值，将参数值重置为 0，并统计 `func` 函数调用次数的方面示例代码如图 4.3 所示。

在一个方面或多个方面代码中，可能存在多个通知的切入点表达式能够与当前连接点匹配。假设当前待匹配的通知序列为 (`act1`, `act2`, ..., `actk`, ..., `actn`)，通知的匹配规则如下所示：

1) 所有的通知按顺序执行匹配处理。

2) 如果当前处理的匹配通知为 `around` 类型通知，且通知的目标代中不包含对 `tjp` 指针 `proceed` 函数的调用，则说明不再存在对原函数的调用，后续的所有待匹配通知都将不再进行匹配处理。

3) 所有的 `before` 和 `after` 位置都是相对于当前原函数的调用位置，如果当前处理的匹配通知为 `around` 类型通知，且通知的目标代码中包含对 `tjp` 指针 `proceed` 函数的调用，则当前对原函

数的调用位置更新为 tjp 指针调用 proceed 函数的位置。

```
monitor statics_monitor(int a) {
    unsigned count = 0;
    pointcut pct1(a) = call(% func(int x:a));
    action act1(*a) before pct1(a) {
        printf("The arg of func is %d\n", *a);
        *a = 0;
        printf("The %d-th execution of func.\n", count);
    }
};
```

图 4.3 MOVEC 方面示例

4.2 语言编译器的设计与实现

MOVEC 语言编译器的输入包括定义了所有横切关注点的方面文件和源 C 语言程序文件，输出是集成了横切关注点功能的目标 C 语言程序文件。MOVEC 语言编译器的结构如图 4.4 所示。

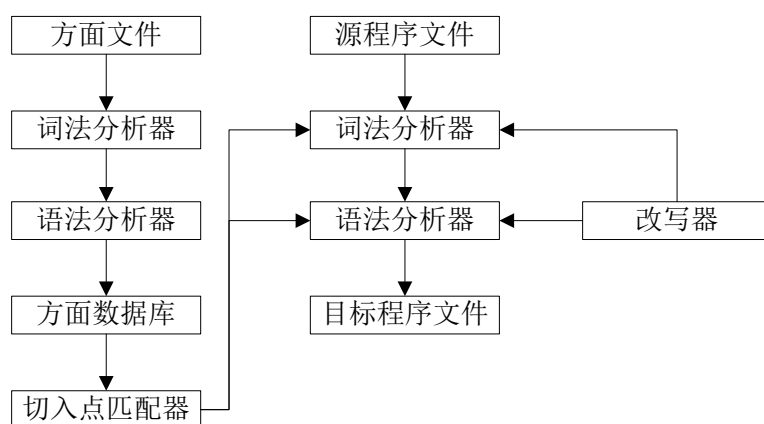


图 4.4 MOVEC 编译器组成结构

MOVEC 语言的编译过程主要包括如下几个步骤：1、利用特定的解析器对方面代码进行词法分析和语法分析，获得其中方面、通知和切入点的定义，存储于特定的数据结构中；2、进行源程序文件的预处理和词法分析，实现宏替换过程，进行文件包含，宏定义等切入点相关的匹配和通知改写处理；3、对于已完成宏替换的程序文件，进行抽象语法树的结构分析，并完成函数，变量等相关切入点的匹配；4、在匹配成功的连接点位置，根据切入点类型的不同，目标动作的不同，切入点和目标动作关联关系的不同，对既有代码进行相应的转换，最后输出目标程序文件。下面将详细描述各个阶段的处理和实现细节。

4.2.1 方面代码解析

MOVEC 语言作为一种新的针对 C 程序的 AOP 语言, 为了实现对切入点, 通知, 方面等概念的描述, 新增了 `monitor`, `pointcut`, `action` 等关键字, 和切入点, 通知, 方面等相关语法结构, 因此词法规则和语法规则会在 C 语言的基础上进行相应的扩充。

在词法分析过程中, 通过 Flex 需要完成识别的词法单元包括如下三类: 1、注释类词法单元, 因为方面文件支持多种格式的注释说明, 注释部分代码将被忽略, 不被语法分析进一步处理; 2、C 语言类词法单元, 包括 C 语言所支持的类型关键字, 操作符关键字, 数字词法单元, 标识符词法单元, 字符串词法单元等, 因为方面文件中支持 C 语言形式的变量声明, 而且方面文件中的切入点, 通知等语法结构中用到了字符串, 标识符等; 3、MOVEC 类词法单元, 主要是方面文件新引入的关键词, 因为方面文件中的切入点, 通知, 方面等语法结构是由关键词和标识符, 字符串共同组合而成。

在语法分析过程中, 按照 MOVEC 设计的语法, 设计 Bison 的语法规则说明, 每条语法规则说明对应特定语法结构的定义, 语法规则中的终结符和非终结符都包含特定的语义值, 而语法规则的附加动作可以用于语法符号语义值的计算, 并将相关信息存储到特定的数据结构中, 作为方面数据库。方面的数据结构设计如图 4.5 所示, 分别与方面, 切入点, 通知语法定义相对应。

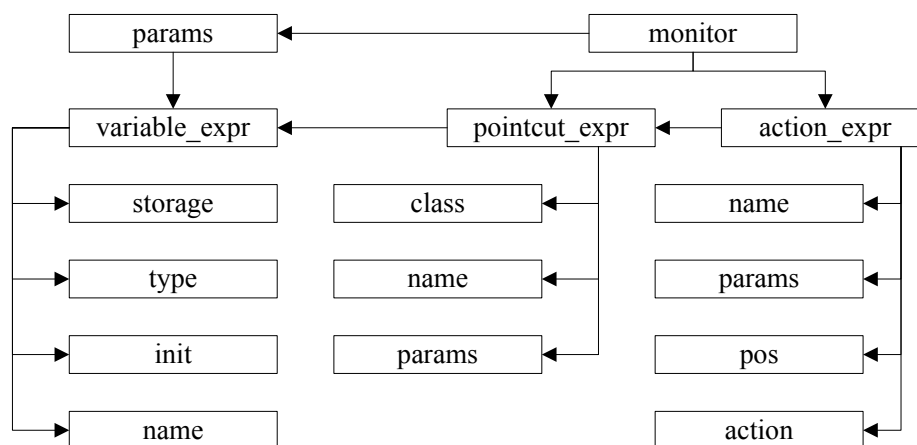


图 4.5 方面数据结构

特别地, 由于基于不同的关键字, 切入点表达式 `pointcut_expr` 会有不同的意义, 进而需要不同的数据结构进行存储, 因此采用继承的方式定义切入点表达式, 切入点表达式继承结构部分如图 4.6 所示。

4.2.2 词法分析与宏替换

正如第二章中编译过程所描述的, 对于输入的源程序文本, 词法分析器将按照 C 语言词法规则将其识别为具有特定意义的字符序列, 即词法单元, 而在 Clang 的设计中, 程序预处理和

词法分析是作为统一整体进行的，预处理过程包含了对包含头文件引入在内的所有预处理指令的执行和宏引用的展开。在这个阶段，主要要完成的工作包括宏替换和预处理指令切入点的匹配处理。

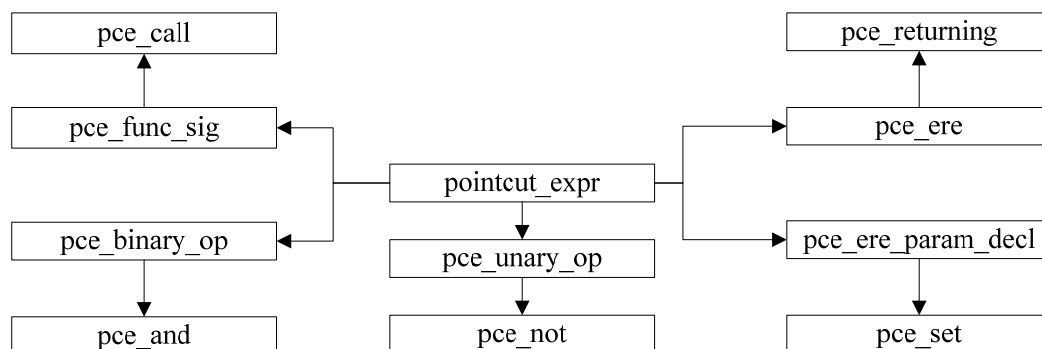


图 4.6 切入点继承结构

4.2.2.1 宏替换的设计实现与优化

对应第二章所描述的编译过程，词法分析是通过读取程序文本，并按照特定的词法规则将其分割为具有特定意义的字符序列，即词法单元。特别地，针对 C 语言的编译，由于存在头文件引入和宏的使用等，在词法分析之前存在一个预处理的过程。预处理过程会将所有的头文件引入指令替换为相应的文件内容，宏也替换为相应的宏定义，在替换结果的基础上进行词法分析。因此，词法分析获得的词法单元中，部分来自于源文件的实际内容，部分来自于预处理引入的头文件，部分来自于宏替换的结果，而这些宏替换的词法单元并不存在实际文件位置，无法直接进行改写。

在部分的 C 语言代码转换工具中，会利用编译器的预处理功能，将所有的源文件进行一次扩展，这样程序不再依赖头文件和宏定义，再在扩展后的代码文件上进行插桩，便不会存在无法改写的问题。但是，这样的处理会改变整个程序的文件结构，使整个程序的代码规模产生膨胀，不利于二次开发，插桩要在尽可能少的改动原有代码的前提下进行。

因此，本文中通过对词法分析过程的改进，在保持程序文件结构不变的前提下，完成宏引用内容的替换，替换过程中需要维护待扩展位置和待扩展内容两个变量，具体算法设计如图 4.7 所示。

根据 Clang 的 LibTooling 编程接口设计，重写其前端处理方法 FrontendAction 可以实现编译过程的自定义，FrontendAction 中提供了一个包含了词法分析器，语法分析器，源代码管理器等各个部分的编译器实例。本文中将继承并重写 PreprocessorFrontendAction 类的 ExecuteAction 方法。Clang 将预处理和词法分析统一在了 lex 模块的 Preprocessor 类中，每调用 Preprocessor 的 Lex 函数一次可以获得一个词法单元 Token。Clang 中所有的位置信息 SourceLocation 由源代码管理器 SourceManager 进行维护，并绑定在词法单元或语法单元的节点

```

while(当前词法单元非空) {
    if(当前词法单元非来自本项目文件) {
        重新获取词法单元, 重新进入循环
    }
    if(当前词法单元来自于文件内容) {
        if(待扩展内容非空) {
            将待扩展内容写回到待扩展位置, 清空待扩展内容
        }
    }
    else if(当前词法单元来自于扩展) {
        if(待扩展内容为空) { 更新待扩展内容, 待扩展位置 }
        else if(待扩展内容非空, 且待扩展位置与当前词法单元扩展位置相同) {
            更新待扩展内容
        } else {
            将待扩展内容写回到待扩展位置
            更新待扩展内容, 待扩展位置
        }
    }
    }
    重新获取词法单元
}
if(带扩展内容非空)
    将待扩展内容写回到待扩展位置

```

图 4.7 宏替换算法

上。通过查询 SourceManger 可以判断当前词法单元位置是否来自于扩展, 如果是, 可以进一步查询获得扩展位置, 扩展的实际内容。最后, 通过 Rewriter 改写器的 ReplaceText 方法实现扩展内容的写回。

对于宏替换的算法, 其中涉及到了当前词法单元是否来自本项目文件的判断, 可以进一步对其进行优化, 加快判断速度。在原有的算法中, 判断逻辑是应用于每一个词法单元, 而词法单元是包含在文件单元中的。词法分析所应用的文件主要包括两类, 一类是本项目文件, 另一类是系统编译器所提供的头文件, 词法分析的过程在遇到 include 指令时会保存当前分析进度并进入新的文件单元, 当一个文件单元分析完成后, 会载入保存的分析进度, 进行未完成的文件单元分析。因此, 可以在每次进入新文件单元或退出旧文件单元的时候, 对当前词法单元进行

分析,判断其是否来自本项目文件,这个判断结果可以应用于后续解析获得的词法单元,直到下一次产生进入新文件单元或退出旧文件单元动作。通过将词法单元级判断调用频率降低为文件单元级判断调用频率,可以大大提高宏替换算法的执行速度。

在 Clang 的预处理模块中,其提供了预处理回调类 `PPCallbacks`,类中函数和事件映射关系包括 `FileChanged` 对应当前处理文件变化, `FileSkipped` 对应文件跳过, `FileNotFound` 对应文件未找到, `moduleImport` 对应模块引入事件等等。通过重写其中的方法,将预处理回调类的实例加入到预处理器, Clang 便会在对文件预处理过程产生事件的时候,自动调用相应的事件处理方法。

通过宏替换的过程,在保持源程序文件结构不变的前提下,实现了宏引用的替换,使得宏中的内容可以被准确的识别和用于后续代码插桩,同时利用判断优化,提升了宏替换的解析速度。

4.2.2.2 预处理指令的匹配

对于预处理指令的匹配,主要是监控程序预处理指令的执行,获取目标引入文件的文件名,目标引用宏名称或目标定义宏名称,通过通配符匹配算法将该文件名或宏名与方面数据库 `include`, `define` 或 `expand` 切入点进行匹配。如果匹配成功,相应通知的目标动作代码将被插入到该预处理指令的相应位置。

特殊的,由于 C 程序中存在宏递归展开,所以在处理宏展开切入点时,需要通过树形结构维护当前宏和其中嵌入宏的结构和展开结果,宏展开切入点的匹配要应用于当前宏和所有嵌入宏,根据匹配结果,将改动写回到树形结构中,最后通过遍历树生成当前宏展开结果,写回到原文件中。

在 Clang 中 `PPCallbacks` 类是一个接口类,其中以虚函数的形式定义了一系列函数,每个函数和预处理过程中的某个事件相对应,例如处理文件发生变化,文件跳过,待处理文件未找到等。如果将 `PPCallbacks` 的对象加入到既有编译器实例中, `PPCallbacks` 中被实现函数都将以回调函数形式,在编译过程中被调用。文件引入预处理指令对应函数 `InclusionDirective`,宏定义对应函数 `MacroDefined`,宏展开对应函数 `MacroExpands`,因此匹配算法和相应代码插桩可以在这些函数的实现中进行。

4.2.3 语法分析与切入点的匹配

Clang 针对词法分析获得的词法单元,会进行相应的语法分析,并构建出抽象语法树。如 2.3 节所述,程序中的声明和语句分别对应抽象语法树中的 `Decl` 类节点和 `Statement` 类节点,同时所有的节点都会包含相关类型信息,位置信息等,利用这些丰富的信息,可以实现抽象语法树节点和切入点的匹配。

参照 3.1.1 节中对切入点语义的定义和 2.3 节中对抽象语法树节点的定义，针对每个基本切入点的匹配都可以映射到抽象语法树的一类特殊类型节点，具体映射如表 4.4 所示。

表 4.4 切入点与语法树节点映射关系

切入点类型	抽象语法树节点类型
infunc	FunctionDecl
intype	TagDecl
infile	TranslationUnitDecl
call	CallExpr
callp	CallExpr
execution	FunctionDecl
set	VarDecl, BinaryOperator 等
get	BinaryOperator, DeclRefExpr 等

正如 2.3 节所述，整个抽象语法树的根节点是 TranslationUnitDecl 类型的抽象语法树节点，Clang 提供了一种对任意节点为根的抽象语法树的递归访问机制，可以实现对任意语法树节点的自定义访问。对于当前正在访问的抽象语法树节点，将对方面数据库中的所有的通知进行遍历，获取其中所引用的切入点表达式，并和当前抽象语法树节点进行匹配，这是切入点匹配的整体流程，下面将就基本切入点匹配和复合切入点匹配两种情况对具体的匹配机制进行介绍。

对于 infunc 切入点，属于范围限制类型切入点，需要将 infunc 切入点所描述的函数返回类型，函数名和函数参数列表同当前语法树节点所在函数的返回类型，函数名和函数参数列表进行通配符匹配，如果匹配成功则说明切入点匹配成功。具体到系统当前语法树节点所在函数的记录，则可以通过 TraverseFunctionDecl 函数实现，因为在语法树的前序遍历中，函数定义节点的访问是先于所有子节点的。

对于 intype 切入点，属于范围限制类型切入点，需要将 intype 切入点所描述的类型名称和当前语法树节点所在类型定义体的名称进行通配符匹配，如果匹配成功则说明切入点匹配成功。需要说明的是，在 C 语言中，不同于函数的定义，结构体的定义是可以嵌套的，所以当前语法树节点所在类型定义体将会以一种栈的形式出现。具体到当前语法树节点所在类型定义栈的记录，则可以通过 TraverseEnumDecl 和 TraverseRecordDecl 函数实现，这两个函数是对枚举类型，结构体类型和联合类型声明的访问。

对于 infile 切入点，属于范围限制类型切入点，需要将 infile 切入点描述的文件名和当前语法树节点所在文件名进行通配符匹配，如果匹配成功则说明切入点匹配成功。具体到当前语法树节点所在文件名，则可以通过 Clang 中的源代码管理器查询获得。

对于 call 切入点，属于动作类型切入点，需要将 call 切入点所描述的函数返回类型，函数

名和函数参数列表同当前语法树节点的返回类型，函数名和函数参数列表进行通配符匹配，如果匹配成功则说明切入点匹配成功。`call` 切入点只对应函数调用 `CallExpr`，因此可以通过 `VisitCallExpr` 函数实现对 `call` 切入点的匹配。

对于 `callp` 切入点，属于动作类型切入点。其与 `call` 切入点的不同之处在于，`callp` 是对函数指针的调用，因此在匹配过程中，需要增加对调用对象为是否指针类型对象的判断，其余部分与 `call` 切入点类似。

对于 `execution` 切入点，属于动作类型切入点，需要将 `execution` 切入点所描述的函数返回类型，函数名和函数参数列表同当前语法树节点的返回类型，函数名和函数参数列表进行通配符匹配，如果匹配成功则说明切入点匹配成功。`execution` 切入点只对应函数定义 `FunctionDecl`，因此可以通过 `VisitFunctionDecl` 函数实现对 `execution` 切入点的匹配。

对于 `set` 切入点，属于动作类型切入点，需要将 `set` 切入点所描述的变量类型和变量名同当前语法树节点的变量的类型和名称进行通配符匹配，如果匹配成功则说明切入点匹配成功。`set` 切入点对应变量声明 `VarDecl`，一元操作符 `UnaryOperator` 等，这些语句节点中可能包含对变量的赋值，对应可以通过 `VisitVarDecl`，`VisitUnaryOperator` 等函数实现对 `set` 切入点的匹配。

对于 `get` 切入点，属于动作类型切入点，和 `set` 同属于变量相关切入点匹配，不同之处在于 `get` 切入点对应二元操作符 `BinaryOperator`，声明引用 `DeclRef` 等，可以通过 `VisitBinaryOperator`，`VisitDeclRef` 等函数实现对 `get` 切入点匹配。

复合操作符作为复合切入点的组成部分，在其定义中是两个连接点集合的运算，因为此处是对单个连接点的匹配，所以将其转换为一种逻辑运算，复合操作符的匹配结果取决于子表达式的匹配结果和当前复合操作符的种类。

`inexec` 切入点和 `returning` 切入点是一类特殊切入点。`inexec` 切入点表示当前语法树节点在指定函数的执行流程中，而函数执行流程的判定属于程序动态状态，无法通过对程序语法结构分析实现。因此在所有包含 `inexec` 切入点的复合切入点匹配计算中，需要将 `inexec` 切入点设置为匹配成功和失败均有可能的情形。`returning` 切入点作为一种辅助命名切入点，并不包含实际匹配含义，同样需要同时考虑匹配成功和失败两种情况。对于包含 `inexec` 切入点和 `returning` 切入点的复合切入点匹配算法如图 4.8 所示。

在具体实现中，由于切入点表达式的数据结构是采用继承方式定义的，如图 4.6 所示，因此对于切入点搜索可以通过在基类 `pointcut_expr` 中增加虚函数，子类中自定义实现的方式完成。对于切入点匹配中的全排列生成，可以通过整数的遍历移位实现，有更高的效率。

切入点匹配算法的输出中，除了匹配结果之外，还输出了赋值方案集合，这里的赋值方案集合代表了程序运行过程中状态的动态约束条件，将在 4.2.4 的代码转换中被使用。

```
1、切入点搜索（输入切入点表达式 pc，输出 inexec 切入点和 returning 切入点集合）
if(切入点表达式为 and, or 切入点) {
    递归搜索左子表达式 l-pc
    递归搜索右子表达式 r-pc
}
if(切入点表达式为 not 切入点)
    递归搜索子表达式 e-pc
if(切入点表达式为 returning 或 inexec)
    将其加入到输出集合中
2、切入点匹配（输入切入点表达式 pc，语法树节点 node，输出匹配结果和赋值方案集合）
搜索当前切入点中 returning 和 inexec 切入点并保存至集合 U
while(针对 U 中的所有切入点的一种全排列赋值) {
    将赋值代入原切入点，按照基本切入点的匹配方法进行计算
    if(匹配成功)
        将此赋值方案存入赋值方案集合中
}
if(赋值方案集合非空)
    匹配成功
else
    匹配失败
```

图 4.8 复合切入点搜索及匹配算法

4.2.4 代码转换

在完成切入点和语法树节点匹配的基础上，根据当前匹配的切入点类型，目标动作，切入点和目标动作关联关系等，参照 C 语言规范，对源代码进行相应的代码转换，目标是通知的目标动作代码能在指定位置与源程序完成横切。

4.2.4.1 静态通知的处理

如 4.1.2 所述，主要是指利用 `extend`，`before` 等关键字对函数结构，文件结构或者结构体类型等进行扩展，或在文件包含指令，宏定义位置进行代码插桩。文件包含和宏相关通知已在词法分析阶段得到处理，因此语法分析阶段主要对函数结构等进行扩展。

如果一个 `extend` 通知匹配成功，则只需要利用 Clang 中的改写器在文件首尾部，函数定义的首部尾部，或结构体的首部尾部插入相应的通知目标代码。

4.2.4.2 切入点上下文数据结构设计

参照 4.1.2 切入点上下文所需要包含的内容，为了兼容 C 语言，切入点上下文采用 struct 结构体进行定义，为支持对 proceed 函数的调用，增加了对原函数的函数地址的保存，具体结构如图 4.9 所示。

```
struct join_point {  
    const char *type;  
    const char *target;  
    const char *target_type;  
    void **args;  
    const char **ags_type;  
    void *ret;  
    const char *file_name;  
    const char *func_name;  
    unsigned loc;  
    void *(*fp)(struct join_point*);  
}
```

图 4.9 接入点上下文数据结构

需要说明的是，为了保持对所有类型的兼容，切入点中地址相关数据采用了空类型设计，而原函数则被处理成为输入参数为切入点上下文，输出返回值地址的函数，这种设计能够解决多通知匹配问题，具体有关多通知匹配和转换方法将在 4.2.4.8 中详细介绍。

4.2.4.3 通知代码转换

方面数据库中需要进行代码转换的主要包含两个部分，变量声明和通知声明。变量声明都将以全局变量的形式出现在方面头文件中，通知声明则将以通知函数的形式出现在方面头文件中。

对于 before 和 after 类型的通知声明，其与通知函数之间是一一对应的关系，因为其函数返回类型只能为 void。而对于 around 类型的通知声明，由于通知函数要替换原有语法树节点的动作，为保证和原程序的执行流兼容，通知函数的返回类型存在非空的可能，也就形成了通知声明和通知函数一对多的关系。具体每个通知函数的返回类型由切入点所匹配的函数返回类型或变量类型决定，通知的参数则按照通知的声明进行设计，另外默认包含一个切入点上下文指针参数，保证通知代码中对 tjp 指针的调用可以正确执行，通知的目标代码中的 proceed 函数调用将被改写为对切入点上下文结构体中 fp 函数指针的调用。

4.2.4.4 inexec 切入点的代码转换

Inexec 切入点是指当前语法树节点在指定的函数执行流中，在代码转换中，每一个 inexec 切入点都将对应一个 inexec 函数，函数中包含一个局部静态变量记录当前状态，在调用 inexec 函数的时候，根据不同的输入参数，改变或返回当前状态。

当对抽象语法树进行遍历时访问到 FunctionDecl 节点，即函数声明节点，且函数的返回类型，函数名和参数列表与 inexec 所描述的函数类型，函数名和参数列表相同，则需要当前函数定义的开始位置和结束位置增加对 inexec 函数的调用，用以改变其中的局部静态变量。Inexec 函数调用插入的方式与 before execution 和 after execution 相同，将在 execution 切入点的代码转换中详述。

4.2.4.5 call 和 callp 切入点的代码转换

Call 切入点是指对函数的调用，假设当前调用函数的原型为 RetType func(t0 p0, tn pn)，调用方式为 value=func(a0, an)，则需要当前语法树节点所在函数体前增加如图 4.10 两个函数，第一个函数是整合了所有通知的函数定义，第二个函数是对原函数的等价表达，其中 before_action_call, around_action_call, after_action_call 分别对应转换后的 before, around, after 通知函数的调用，struct join_point 为连接点上下文结构体，inexec_cond 为 inexec 切入点引入的动态判定条件。

需要说明的是，inexec_cond 动态判定条件来自于切入点匹配过程中的赋值方案集合，每一个赋值方案集合都对应一组 inexec 切入点表达式，而在 4.4.2 节中，每一个切入点表达式都被转换为一个 inexec 函数获取状态，因此通过如图 4.11 方式构成动态判定条件。

最后将原 func(a0, an)调用改写为 PRF_func_SID(a0, an)的调用，这样原有对 func 函数的调用将重新指向对 PRF_func_SID 函数的调用，实现在 func 函数的调用前后和过程中执行相应通知目标动作。

Callp 切入点是指对函数指针的调用，对于(*func_p)(a0, an)函数指针调用，可以将函数指针作为参数进行传递，改写为 func_call(a0, an, *func_p)，这样便可以采用和 call 类似的代码转换方式。

4.2.4.6 execution 切入点的代码转换

Execution 切入点是指函数的执行，假设当前声明函数的原型为 RetType func(t0 p0, tn pn)，则需要当前语法树节点所在函数体前增加如图 4.12 两个函数，第一个函数是整合了所有通知的函数定义，第二个函数是对原函数的等价表达，其中 before_action_call, around_action_call, after_action_call 分别对应转换后的 before, around, after 通知函数的调用，struct join_point 为连


```

RetType PRF_func_SID(t0 p0, tn pn) {
    RetType ret_value;    //返回值变量声明
    struct join_point tjp; //切入点上下文声明
    tjp.fp = &PRF_func_SID_wrapper_1; //切入点上下文初始化
    .....
    if(inexec_cond) before_action_call(&tjp, p0, pn, ret_value); //before 通知
    if(inexec_cond) //around 通知
        ret_value = around_action_call(&tjp, p0, pn, ret_value);
    else
        ret_value = *(RetType*) PRF_func_SID_wrapper_1 (&tjp);
    if(inexec_cond) after_action_call(&tjp, p0, pn, ret_value); //after 通知
    return ret_value;
}

void* PRF_func_SID_wrapper_1(struct join_point *tjp) {
    RetType *ret_value = (RetType*)tjp->ret; //从切入点中取出返回值，参数等
    .....
    *ret_value = func(*p0, *pn); //原函数调用
    return ret_value;
}

```

图 4.10 call && callp 代码转换规则

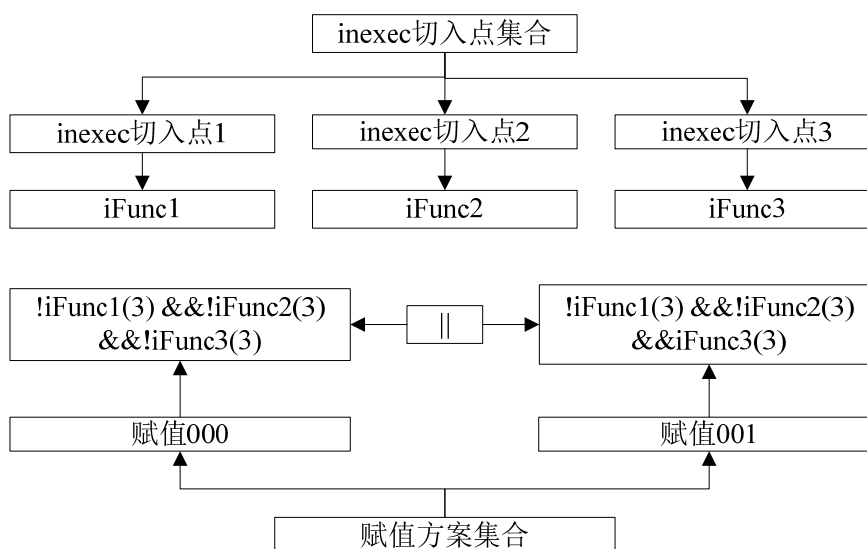


图 4.11 inexec 动态判定条件生成算法

接点上下文结构体，inexec_cond 为 inexec 切入点引入的动态判定条件。

最后将原有声明 `RetType func(t0 p0, tn pn)` 改写为 `RetType PRF_func(t0 p0, tn pn)` 的声明，这样对原有 `func` 函数的执行都将重新指向新增加的第一个函数，实现在 `func` 函数的执行前后和过程中执行相应通知目标动作。

```
RetType func(t0 p0, tn pn) {
    RetType ret_value;    //返回值变量声明
    struct join_point tjp; //切入点上下文声明
    tjp.fp = &PRF_func_SID_wrapper_1; //切入点上下文初始化
    .....
    if(inexec_cond) before_action_call(&tjp, p0, pn, ret_value); //before 通知
    if(inexec_cond) //around 通知
        ret_value = around_action_call(&tjp, p0, pn, ret_value);
    else
        ret_value = *(RetType*) PRF_func_SID_wrapper_1 (&tjp);
    if(inexec_cond) after_action_call(&tjp, p0, pn, ret_value); //after 通知
    return ret_value;
}

void* PRF_func_SID_wrapper_1(struct join_point *tjp) {
    RetType *ret_value = (RetType*)tjp->ret; //从切入点中取出返回值，参数等
    .....
    *ret_value = PRF_func(*p0, *pn); //原函数调用
    return ret_value;
}
```

图 4.12 execution 代码转换规则

4.2.4.7 set 和 get 切入点的代码转换

Set 切入点是指对变量的赋值操作，get 切入点是指对变量的取值操作。通过分析，Clang 抽象语法树中的 `UnaryOperator`（一元）操作符节点中可能包含对变量的赋值和取值，例如 `a++` 和 `a--`，`CompoundAssignOperator`（复合赋值）操作符节点可能包含对变量的赋值和取值，例如 `a+=2` 和 `a-=2`，`VarDecl`（变量声明）节点可能包含对变量的赋值，例如 `int a = 2`，`BinaryOperator`（二元）操作符节点中的等号操作符节点可能包含对变量的赋值，`DeclRefExpr`（声明引用）节点中可能包含对变量的取值。

所有对变量的赋值可以理解为对赋值函数 `set_var` 的调用，对变量的取值可以理解为对取值函数 `get_var` 的调用，这样 set 和 get 切入点的处理就可以转化为类似 call 切入点的处理。不同

情形下变量赋值取值与函数调用的转换原则如图 4.13 所示。

//一元操作		
++var	=>	set_var(&var, get_var(&var) + 1)
var++	=>	set_varPP(&var, get_var(&var) + 1)
//复合赋值操作		
var+=a	=>	set_var(&var, get_var(&var) + a)
//变量声明		
Type var = a	=>	Type var = (PRF_tmp_1 = a, set_var(&PRF_tmp_1, PRF_tmp_1));
//二元操作		
var = a	=>	set_var(&var, a)
//声明引用		
var	=>	get_var(&var)

图 4.13 变量赋值取值与函数调用等价转换

如果出现特殊情形，转换规则需要特殊处理。例如，当变量声明为 **register** 类型或数组类型时，因为无法直接获取地址，需要增加临时变量对原变量进行存储，在临时变量上进行操作并最终把临时变量赋值给原变量。

经过与 call 切入点类似的代码转换后，原有的取值赋值操作，重新指向对 **get_var** 或 **set_var** 函数的调用，实现在变量取值赋值的前后和过程中执行相应通知目标动作。

4.2.4.8 多通知的代码处理

因为在同一个语法树节点，可能存在多个通知与之相匹配，该语法树节点的代码转换设计需要保证满足如下两个要求，一是与 C 语言标准相符合，二是满足 4.1.2 中对于多通知匹配关于优先级的定义。下面以两个包含 **proceed** 调用的 **around** 类型通知同时匹配一个语法树节点为例，说明多通知匹配的代码设计。如图 4.14 所示，是按照多通知匹配定义所设计的多通知匹配代码处理方式。

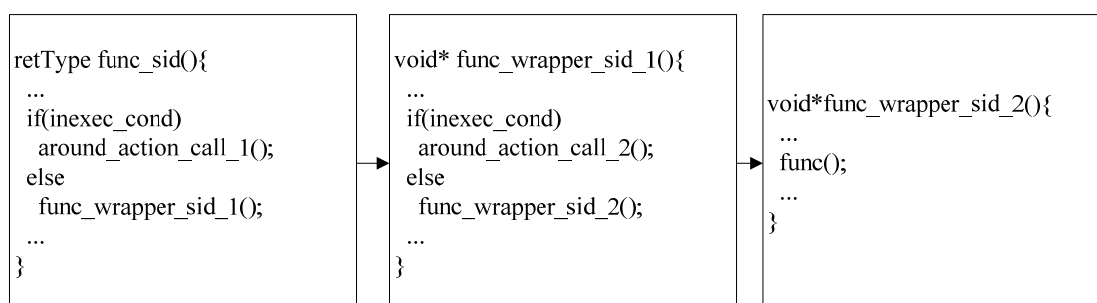


图 4.14 多通知匹配代码逻辑

如果按照传统意义上的处理方式，当第一个 **around** 类型通知匹配并处理完毕时，对原函数

的调用位置包含两处，一处位于函数 `func_sid` 的 `else` 分支，另一处位于 `around` 通知 `proceed` 函数所调用的 `func_wrapper_sid_1` 中。当第二个 `around` 类型通知需要匹配并处理时，需要同时对两处位置进行代码处理，这种方式会使代码调用关系管理变得复杂。

通过分析可以发现，在第一个 `around` 类型通知匹配完毕后，函数 `func_sid` 的 `else` 分支和 `func_wrapper_sid_1` 函数本质上都是对原函数的调用，通过将函数 `func_sid` 的 `else` 分支中对 `func` 函数的调用，改为对 `func_wrapper_sid_1` 函数的调用，消除了第一个原函数调用位置，后续的匹配只需要作用于调用关系链的最后一个函数，通过这种优化的方式，简化了代码调用关系的管理。

4.2.5 单文件编译与项目编译

MOVEC 语言编译器支持两种工作方式，一种是对单文件的编译，另一种是对项目文件夹的编译。

对于任一种工作方式，相应的编译结果，即目标程序文件或目标项目文件夹，都需要输出到用户指定的位置。因此，作为 MOVEC 语言编译器功能的一部分，需要基于不同的文件系统实现对文件相关操作的支持。目前 MOVEC 语言编译器实现了如下文件相关操作，当前文件路径取得，绝对路径计算，相对路径计算，父文件夹计算，文件名计算，文件类型计算，文件路径合成，文件和文件夹的创建与删除，特定类型，前缀，后缀文件的搜索。所有这些文件操作，均分别在 Linux 平台和 Windows 平台进行了实现。

特别地，不同于单文件编译可以在 MOVEC 编译器调用时直接指定相关编译指令以实现词法和语法分析，一个项目的编译可能包含很多的源文件，为了提高 MOVEC 编译器的实用性，MOVEC 提供了一种 json 文件的解析功能，json 文件中存储项目中的源文件路径和对应编译指令，可以用于 MOVEC 编译器的调用。而该 json 文件可以通过对项目实际编译过程中 C 语言编译器调用的监控和记录生成。

4.3 本章小结

本章首先介绍了 MOVEC 语言的语法设计，在切入点描述方面，和 ACC 相比较，增加了文件包含切入点，宏定义和宏展开切入点，和 AC++相比较，增加了文件包含切入点，宏定义和宏展开切入点，变量赋值和取值切入点，同时取消了 ACC 和 AC++中的 `args` 切入点，将其功能融合到 `call` 等既有切入点中，增加了函数参数重命名功能，使语言更加简洁易用；在通知描述方面，和 ACC 相比，不再要求写明通知的返回类型和参数类型，而是通过后续代码匹配的结果自动判定；在方面描述方面，和 ACC 相比，利用 `monitor` 关键字实现了不同横切关注点的分离与模块化。

然后介绍了 MOVEC 语言的实现，和 ACC 和 AC++相比较，在方面文件解析方面，通过 `flex`

和 `bison` 工具的使用，使 `MOVEC` 在语法规则上更易于扩展；在预处理方面，通过部分替换，既维持了程序原有文件结构，避免头文件带来的代码冗余，又实现了文件包含切入点，宏定义和宏展开切入点的处理；在切入点匹配方面，`Clang` 使抽象语法树的生成匹配更加快速；在代码转换方面，更多的考虑到了前置后置一元操作符，`register` 类型等特殊情况的处理，保证其正确性，而且代码转换的设计使多个通知应用在同一个接入点时不会产生冲突。

第五章 C 语言常见错误的运行时验证插桩

5.1 常见软件错误定义及相关研究

在 C 语言编程中, 存在一些软件常见错误, 下面将就除 0, 整数溢出, 未初始化变量引用三种错误介绍其定义及相关研究。

5.1.1 C 语言及其常见错误定义

软件失效是指计算机得到的、观察到的、测量到的数值或条件和系统设计所要求的数值或者条件之间存在差异, 软件缺陷是指系统当前的行为和期望的行为之间存在差异, 通常可以通过将系统当前运行状态和期望的运行状态进行对比达到识别的目的, 软件错误是一种人为引入系统, 成为缺陷, 并最终有可能导致软件失效的行为^[52]。

自从 1972 年美国贝尔实验室的 Ritchie 在 B 语言的基础上重新设计出 C 语言, C 语言已经发展和应用了 40 多年。根据 TIOBE 最新的编程语言排行榜, C 语言在使用率上仅次于 Java 语言, 所占比率均在 15% 以上。虽然, Java 在 Web 应用方面具有跨平台的特定, 但 C 语言有其特殊的优势: 1) 相比较于 Java 的解释执行, C 语言的编译执行具有更高的时间效率; 2) 相比较于 Java 的自动内存控制和虚拟机技术, C 语言的手动内存控制具有更好的空间利用率。因此, 在时间控制要求高的实时系统, 内存控制要求高的嵌入式系统, 需要直接访问硬件的驱动程序等开发当中, C 语言都会是更好的选择。

C 语言非常灵活, 在编程设计方面给了程序员的操作选择空间, 但同时由于 C 语言不是一种强类型语言, 在语法限制方面不严格, 缺少类型约束, 运行时错误检查功能也很少。这就导致 C 语言编程中会出现一些错误, 这些错误可能来自于程序员的设计错误, 对语言标准的理解错误, 对编译器实现的理解错误等, 使系统产生缺陷, 并最终可能导致系统的失效, 例如除 0 错误, 整数溢出错误, 使用未初始化变量错误等等。

在 ISO C99 语言标准的定义中, 如果除法运算的第二个操作数为 0, 除法运算的行为将是未定义的, 而在软件的设计上, 软件的可靠运行是不能依赖于未定义行为的。根据 US-CERT 的漏洞数据库, 除 0 错误曾经导致 NetBSD 和 LibTIFF 的拒绝服务漏洞。

对于一个特定的计算机平台, 其基本类型都有一个固定的宽度。例如对于 64 位 Windows 系统 VS 编译器中, char 类型占用空间为 1 个字节, int 类型占用空间为 4 个字节, long 类型占用空间为 4 个字节。固定的空间, 使得这些基本类型在用来表示整数的时候, 存在一个类型表达能力上限和下限的问题, 一旦当变量所要保存的值超出了其类型上限或类型下限, 就会导致整数溢出。根据 ISO C99 语言标准的定义, 整数溢出错误同样属于未定义行为, 尤其是在与其

他操作相结合之后,可能产生堆溢出和栈溢出等其他更加严重的错误。在1980年发布的Pac-Man吃豆人程序中,两个整数溢出漏洞导致了用户画面出现错误。根据NVD(National Vulnerability Database)的统计,整数溢出漏洞广泛存在于Adobe Flash Player, Microsoft Internet Explorer, Google Chrome当中,并都有最高的危险等级。

在C语言中,全局变量和静态变量会由编译器自动完成初始化,默认值为0,而其他的变量如果未经初始化,变量值由变量声明时对应地址块的内容决定,具有不确定性,有可能导致系统崩溃,而且不易于验证。

5.1.2 整数溢出错误验证插桩相关研究

因为在三种常见错误中,整数溢出错误对系统有相当的可靠性影响,而其相关研究也相对更为丰富,所以这里主要总结整数溢出错误验证插桩的相关研究。

为了解决C语言程序中存在的整数溢出漏洞,目前开发出了一系列的C语言变种语言,例如CCured和Cyclone,利用这些变种语言进行编程就可以保证不再产生溢出,但这与自动化验证整数溢出的初衷背离,将原始程序利用变种语言重新进行编写需要人工的参与,不符合自动化验证插桩的要求。

对于整数溢出验证的研究主要可以从两个角度进行分类,一个角度是验证的时间,包括静态分析和运行时验证,另一个角度是验证的对象,包括源代码级验证,中间代码级和二进制代码级。下面将主要从验证时间的角度分类介绍目前已有的一些验证工具及其实现。

首先对于运行时验证,IOC是一个动态验证工具,作为一种Clang编译器前端的扩展,其采用的是基于AST进行中间代码插桩的方法进行运行时验证^[53]。Clang在经过词法分析和语法分析后会生成抽象语法树AST,然后通过CodeGen模块将抽象语法树转换为中间代码,IOC即是在转换过程插入相关代码,中间LLVM IR代码将最终通过后端生成可执行文件。IOC支持的验证逻辑包括两部分,一是采用先决条件校验,例如有符号a+b溢出必须满足先决条件((a > 0) && (b > 0) && a > (INT_MAX - b)) || ((a < 0) && (b < 0) && a < (INT_MIN - b)),二是采用CPU符号位进行测试,LLVM中提供了内在函数以访问该CPU符号位。

RICH同样是一个动态验证工具,作为一种Gcc编译器的扩展,其采用的是基于Gcc IR中间代码进行目标代码插桩的方法进行运行时验证^[54]。Gcc在经过词法分析,语法分析等过程会生成中间代码,然后通过目标代码生成模块将中间代码转换为目标代码,RICH即是在转换过程插入相关指令,目标代码最终通过链接生成可执行文件。RICH支持的验证逻辑包括两个方面,一是截断和类型转换的改写,例如uint32_t变量a赋值给uint16_t变量b将增加如(a <= 216 - 1) && (a >= -216)的判断逻辑,二是溢出操作的改写,例如uint8_t变量a和uint8_t变量b相加,会将a和b转换为uint16_t,对相加结果res增加判断(res <= 27 - 1) && (res >= -27)。

PICK 是 RICH 相应的一个基于源代码插桩版本的运行时验证, PICK 的实现是基于 CIL 这个 C 语言分析和源代码转换框架, PICK 只实现了对于类型转换的溢出验证^[55]。

BRICK 也是一个动态验证工具, 通过修改 Valgrind 动态二进制插桩框架并利用其类型接口插件 Catchconv 对编译生成的可执行文件插桩的方法进行运行时验证^[56], 缺点是代码执行效率低且缺少 C 语言级别类型信息容易导致漏报和误报。BRICK 的执行过程主要包括三部分, 一是将二进制代码转换为 Valgrind 的中间代码 VEX, 二是在整数相关语句执行中记录相关信息, 利用已插入的验证逻辑进行溢出验证和定位。BRICK 的验证逻辑类似于 IOC 的先决条件校验, 这里不再详述。

SmartFuzz 与 BRICK 同样是基于修改的 Valgrind 动态二进制插桩框架, 不同之处在于其增加了通过白盒测试方法生成测试用例, 增加了测试覆盖率^[57]。

对于静态分析, IntScope 是一个静态二进制分析工具, 其将反汇编代码转换为自定义的中间代码格式, 并在这些中间代码的基础上进行符号执行和污点分析, 用以验证整数漏洞的存在^[58]。ASTREE 是一个抽象化的基于解释的静态程序分析器, 目的是实现 C 语言代码的无运行时错误的证明^[59]。因为静态分析不是本文研究的重点, 所以这里不再详述。

综合上述, 静态分析由于本身无法体现程序运行时信息的限制, 只能验证出潜在的整数溢出, 具有较高比率的误报和错报, 更重要的是静态分析无法解决有符号错误和赋值截断带来的溢出错误。静态分析对于漏洞检出方面始终无法给出确定的保证。

相比较而言, 动态分析能够通过验证代码插桩充分利用程序运行时的变量实际值, 目前的工具主要基于二进制代码和中间代码插桩的研究, 基于源代码插桩的 PICK 也仅实现了类型转换的溢出验证。

源代码插桩的形式和二进制及中间代码插桩形式相比较, 一方面可以获得更多的变量类型信息和代码结构信息, 例如在二进制文件中只有在比较相关语句中才包含类型信息, 而源文件解析获得抽象语法树可以获得任意变量的类型信息, 辅助整数溢出验证代码的生成, 另一方面, 源代码插桩后的代码可以进行二次开发和跨平台重复利用, 跨平台的特性可以使包含整数溢出验证的程序代码在任意平台用任意兼容 C 的编译器进行编译运行。

所以本章剩余部分中将设计采用源代码插桩的形式, 实现类型转换和运算所可能带来的整数溢出错误的验证插桩。

5.2 常见错误的验证插桩设计与实现

本节将就上述三种常见错误, 设计其验证插桩逻辑及 Clang 编译器插桩实现。

5.2.1 除 0 错误的验证插桩设计与实现

按照除 0 错误的定义, 除 0 错误发生在除法运算和复合除法赋值运算中, 所以可以通过引

入函数对除法运算和复合除法赋值运算的第二个操作数进行实时检查，如果发现除数为 0 则立即报告错误所在文件位置。

对于除法运算和复合除法赋值运算，其相应的代码转换规则如图 5.1 所示，其中 `type` 为表达式 `b` 的类型，`check_zero` 为引入的验证函数，`file_path` 为运算所在文件路径，`line_no` 为运算所在代码行号。

```
a / b -> a / ((type)check_zero(b, file_path, line_no))
a /= b -> a / ((type)check_zero(b, file_path, line_no))
```

图 5.1 除 0 验证代码转换规则

代码转换规则中的 `check_zero` 函数的实现如图 5.2 所示，判断第一个参数是否为 0，如果为 0 则表示发生了除 0 错误，打印输出运算发生所在文件和代码行位置，否则返回第一个参数。因为浮点数存在不精确的问题，所以对于参数为 0 的判断，需要对浮点数取绝对值并判断其在一定的精度范围内。

```
long double check_zero(long double num, const char* file_path, unsigned line_no) {
    if(fabs(num) < 1e-6)
        printf("Divided by zero file : %s line: %d.\n", file_path, line_no);
    return num;
}
```

图 5.2 除 0 验证判断函数

在 Clang 中，除法运算对应的抽象语法树节点为 `BinaryOperator` 类型节点，通过重写 `RecursiveASTVisitor` 的 `VisitBinDiv` 方法可以实现对该节点的访问；复合除法赋值运算对应的抽象语法树节点为 `CompoundAssignOperator` 类型节点，通过重写 `RecursiveASTVisitor` 的 `VisitBinDivAssign` 方法可以实现对该节点的访问。

5.2.2 整数溢出的验证插桩设计与实现

根据 ISO C99 语言标准，对所有的运算进行分类统计可以得知，整数溢出可能发生在二元操作符，二元操作符，复合赋值操作符，类型转换这四大类运算过程中。溢出验证可以在返回运算结果之前，通过对各类运算的操作数进行类型扩展后再计算，比较两次计算结果，如果不同则立即报告错误所在位置。特殊地，对于整数移位操作的溢出操作，其溢出验证可以简化为判断移位的位数是否超过操作数类型的位宽。

在代码转换规则的设计中，验证代码要保证不改变源程序的原有执行流程和执行结果。在这里主要考虑两个方面的问题，一是代码执行可能存在副作用，因此转换后代码执行次数不能发生变化，例如 `(a++, b)` 这样的代码虽然始终返回变量 `b` 的值，但是存在修改变量 `a` 的副作用；二是代码执行可能存在相互依赖，因此转换后代码执行的顺序不能发生变化，例如在 C 语言中

存在许多特别的运算顺序设计，例如 $f((a, b), (c, d))$ 中的计算顺序是 $cabd$ ，而 $(a, b) + (c, d)$ 中的计算顺序是 $acbd$ ，这主要是由于逗号表达式，函数调用和加法运算在操作数运算顺序上的特殊形成的。下面将分别针对不同操作符的代码转换规则，介绍其在执行次数和执行顺序相关的设计。

另外需要说明的是，因为在实际程序中这四类运算均存在嵌套的可能，Clang 提供了语法树节点遍历机制是先序遍历，从效果上就是先访问运算本身，再访问运算的操作数。然而 Clang 中的改写器 Rewriter 对于同一个位置多次插入内容的规则是，如果是在一个位置前插入内容，后插入的内容在先插入内容的首部，如果是在一个位置后插入内容，则后插入的内容在先插入内容的尾部。为了避免运算嵌套带来的插入代码交叉，需要重新设计一个语法树节点遍历类，采用后序遍历机制，先对操作数进行分析插桩，再对运算本身进行分析插桩，这样便不会产生插桩代码的交叉，由于语法树节点遍历相对较复杂，所以针对后序遍历的修改不再详细描述。

5.2.2.1 一元操作的设计与实现

可能产生整数溢出的一元操作包括取反，前置加法，前置减法，后置加法，后置减法。对于一元操作符，代码插桩规则如图 5.3 所示，其中 `type` 为整体表达式的类型，`type_sub` 为表达式 `a` 的类型，`over_uo` 为引入的验证函数，`temp_1` 为 128 位整数临时变量，`temp_p` 为 128 位临时指针变量，`file_path` 为运算所在文件路径，`line_no` 为运算所在代码行号，`kind` 代表运算所属一元操作符种类。

```
//一元操作符
取反 -a ->
((type)over_uo(-(type_sub)temp_1, -temp_1, file_path, line_no, kind, temp_1 = (a)))
前置加 ++a ->
((type)over_uo(++(*(type_sub*)temp_p),  *(type_sub*)temp_p  +  (__int128_t)1,  file_path,
line_no, kind, temp_p = &(a)))
```

图 5.3 一元操作代码转换规则

对于一元操作来说，原有的执行流程可以分解为计算操作数，计算一元操作结果，返回一元操作结果，在引入了验证函数之后，执行流程便分解为计算操作数，计算类型扩展后一元操作结果，计算一元操作结果，返回一元操作结果。一方面因为表达式 `a` 的计算可能存在副作用，所以为了保证执行次数正确性，表达式 `a` 的计算结果需要用临时变量 `temp_1` 或 `temp_p` 进行存储。另一方面，因为程序中运算可能存在嵌套，而 `temp_1` 和 `temp_p` 临时变量为整个程序共用，所以需要采用一种语法结构保证四个执行动作的连续性和先后顺序，同时这种语法结构不会产生副作用，例如 $(a, b) + (c, d)$ 中就破坏了 `ab` 执行动作的连续性，所以在这里采用函数调用的语法结构。函数调用中参数的计算是按照从右向左依次计算的，且严格保证参数计算的连续性，函数的返回动作可以进行自定义，例如根据参数值判断是否存在溢出错误，并返回一元操作结

果。

代码转换规则中的 `over_uo` 函数的实现如图 5.4 所示,判断第一个参数和第二个参数是否相等,如果不等则表示发生了溢出错误,打印输出运算发生所在文件和代码行位置,否则根据一元操作类型和参数一,重新计算一元操作的实际返回值,并将最终结果进行返回。

在 Clang 中,一元操作符对应的抽象语法树节点为 `UnaryOperator` 类型节点,通过重写 `RecursiveASTVisitor` 的 `VisitUnaryOperator` 方法可以实现对所有一元操作符节点的访问,一元操作的操作数作为一元操作符的子节点存在。

```
__int128_t over_uo(__int128_t num1, __int128_t num2, const char* file_path, unsigned line_no,
unsigned kind, __int128_t addition) {
    if(num1 != num2)
        printf("Divided by zero file : %s line: %d.\n", file_path, line_no);
    if(kind == 1) return num1-1;    //后置加 a++
    else if(kind == 2) return num1+1; //后置减 a--
    else return num1;    //其他类型
}
```

图 5.4 一元操作溢出验证判断函数

5.2.2.2 二元操作的设计与实现

二元操作包括两种类型,一种为简单二元操作,加法,减法,乘法,除法,左移和右移,另一种为复合赋值二元操作,复合加赋值,复合减赋值,复合乘赋值,复合除赋值,复合左移赋值,复合右移赋值,下面分别就两种类型进行讨论。

对于简单二元操作符,代码插桩规则如图 5.5 所示,其中 `type`, `type_l` 和 `type_r` 为整体表达式, `a` 和 `b` 的类型, `over_bo` 和 `over_sh` 为引入的验证函数, `temp_1` 和 `temp_2` 为 128 位整数临时变量, `file_path` 为运算所在文件路径, `line_no` 为运算所在代码行号。

```
//二元操作符
加法 a+b ->
(((type)over_bo((type_l)temp_1 + (type_r)temp_2, temp_1 + temp_2, file_path, line_no, (temp_1
= (a), stack_push(temp_1), temp_2=(b), temp_1=stack_pop(), 0)))
左移 a<<b ->
a<<(((type_r)over_sh(temp_1, sizeof(type_l), file_path, line_no, temp_1=(b)))
```

图 5.5 简单二元操作代码转换规则

因为表达式 `a` 和表达式 `b` 中均可能存在溢出验证的嵌套,而临时变量 `temp_1` 为全局共用,所以需要设计了一个整数栈和相关函数作为辅助机制,以解决表达式 `b` 计算过程中对 `temp_1`

可能的覆盖。整数栈采用链表的形式进行存储，`stack_push` 表示入栈，`stack_pop` 表示出栈，`stack_clear` 表示清空栈内容。同时为了保证表达式 `a` 和表达式 `b` 计算的先后顺序，将 `temp_1` 和 `temp_2` 的计算部分单独用逗号表达式进行组织。

代码转换规则中 `over_bo` 函数的实现如图 5.6 所示，判断第一个参数和第二个参数是否相等，如果不等则表示发生了溢出错误，打印输出运算发生所在文件和代码行位置，否则根据参数一，返回二元操作的结果。

```
__int128_t over_bo(__int128_t num1, __int128_t num2, const char* file_path, unsigned line_no,
__int128_t addition) {
    if(num1 != num2)
        printf("Divided by zero file : %s line: %d.\n", file_path, line_no);
    return num1;
}
```

图 5.6 简单二元操作溢出验证判断函数

在 Clang 中，简单二元操作对应的抽象语法树节点为 `BinaryOperator` 类型节点，通过重写 `RecursiveASTVisitor` 的 `VisitBinaryOperator` 方法可以实现对所有简单二元操作符节点的访问，简单二元操作的操作数作为二元操作符的子节点存在。

对于复合赋值二元操作符，代码插桩规则如图 5.7 所示，其中 `type`，`type_l` 和 `type_r` 为整体表达式，`a` 和 `b` 的类型，`over_bo` 和 `over_sh` 为引入的验证函数，`temp_1` 为 128 位整数临时变量，`temp_p` 为 128 位整数临时指针变量，`file_path` 为运算所在文件路径，`line_no` 为运算所在代码行号。

复合赋值二元操作符的代码转换逻辑与简单二元操作符基本类似，这里不再详述，唯一不同的是，这里将第一个操作数的地址作为整数进行了压栈存储。

```
//复合赋值操作符
复合加赋值 a+=b ->
((type)over_bo(*(type_l*)temp_p) += (type_r)temp_1, (*(type_l*)temp_p) + temp_1, file_path,
line_no, (temp_p = &(a), stack_push(temp_1), temp_2=(b), temp_1=stack_pop(), 0)))
复合左移赋值 a<<=b ->
a<<=((type_r)over_sh(temp_1, sizeof(type_l), file_path, line_no, temp_1=(b)))
```

图 5.7 复合赋值二元操作代码转换规则

在 Clang 中，复合赋值二元操作符对应的抽象语法树节点为 `CompoundAssignOperator` 类型节点，通过重写 `RecursiveASTVisitor` 的 `VisitCompoundAssignOperator` 方法可以实现对所有复合赋值二元操作符节点的访问，复合赋值二元操作的操作数作为复合赋值二元操作符的子节点存在。

5.2.2.3 类型转换的设计与实现

类型转换主要包括显式类型转换和隐含类型转换，对于显式类型转换，如果发生整数溢出，可以理解为程序编写者的特别设计，然而对于隐含类型转换，则无法判断是否在程序编写者的设计范围内，需要对其进行整数溢出验证。

对于隐含类型转换操作符，代码插桩规则如图 5.8 所示，其中 `type` 为表达式 `a` 目标转换类型，`over_bo` 为引入的验证函数，`temp_1` 为 128 位整数临时变量，`file_path` 为运算所在文件路径，`line_no` 为运算所在代码行号。

```
//类型转换  
隐含类型转换 a ->  
((type)over_bo((type)temp_1, temp_1, file_path, line_no, temp_1=(a)))
```

图 5.8 类型转换代码转换规则

代码转换规则中 `over_bo` 函数的实现与二元操作符代码转换规则中所使用的 `over_bo` 函数实现相同，判断第一个参数和第二个参数是否相等，如果不等则表示发生了溢出错误，打印输出转换发生所在文件和代码行位置，否则根据参数一，返回转换操作的结果。

在 Clang 中，隐含类型转换操作符对应的抽象语法树节点为 `ImplicitCastExpr` 类型节点，通过重写 `RecursiveASTVisitor` 的 `VisitImplicitCastExpr` 方法可以实现对所有隐含类型转换操作符节点的访问，隐含类型转换操作的操作数作为隐含类型转换操作符的子节点存在。

5.2.3 未初始化变量引用的验证插桩设计与实现

根据 ISO C99 语言标准，在 C 程序中声明的变量类型中，全局变量和静态变量的内存空间会被自动置 0，因此对于未初始化变量的验证主要关注的是局部变量的声明，赋值和使用。在变量声明和赋值的时候更新变量初始化状态，在变量使用的时候检查变量初始化状态，如果变量依然为未初始化，则表示引用了未初始化变量，立即报告错误所在位置。

为了进一步降低插桩代码的数量，采用集合记录待检测的变量，如果变量在声明时包含初始化表达式，则不加入待检测变量集合中，具体搜索验证算法如图 5.9。通过对声明语句的分析获得待检测变量集合，通过对类型转换节点，一元操作符节点和二元操作符节点的分析获得变量值使用位置，通过对变量引用节点分析获得变量赋值位置。一元操作符和二元操作符节点属于特殊类型节点，既包括变量取值，也包括变量赋值，所以需要作为变量取值进行特殊处理。需要说明的是，变量引用节点是作为类型转换，一元操作符，二元操作符等节点的子节点存在的，根据算法的要求必须先访问类型转换，一元操作符，二元操作符节点，再访问变量引用子节点，所以在引用未初始化变量的验证中，选择使用语法树节点的先序遍历类，而不再是整数溢出验证中所使用的后序遍历类。

对于变量值使用的插桩规则 A 和变量赋值的转换规则 B 如图 5.10 所示，其中 `check_uninit` 函数的参数 `array_k` 为当前编译函数单元用于记录变量初始化状态的数组，`index` 为变量 `a` 在待检测变量集中的序号，`file_path` 为变量 `a` 所在文件位置，`line_no` 为变量 `a` 所在的文件行号。

```
while(当前语法树节点非空) {
    if(当前语法树节点为声明语句) {
        if(当前声明语句为全局声明或静态声明)
            跳过
        while(当前声明非空) {
            if(当前声明为变量声明且没有初始化表达式)
                将该变量声明加入待检测变量集合 Q 中
            获取下一个声明
        }
    }
    if(当前语法树节点为左值向右值的类型转换)
        if(类型转换对象为变量声明引用且该变量声明在待检测变量集合 Q 中)
            该处为变量值使用，执行插桩规则 A
    if(当前语法树节点为一元前置后置操作符或二元复合赋值操作符)
        if(操作数对象为变量声明引用且该变量声明在待检测变量集合 Q 中)
            该处为变量值使用，执行插桩规则 A
    if(当前语法树节点为声明引用) {
        if(当前声明引用已被作为类型转换，一元前置后置操作符或二元复合赋值操作符处理)
            跳过不处理
        if(当前声明引用为变量声明引用且该变量声明在待检测变量集合 Q 中)
            该处为变量赋值，执行插桩规则 B
    }
}
```

图 5.9 未初始化变量引用验证算法

`set_uninit` 函数的参数 `array_k` 为当前编译单元用于记录变量初始化状态的数组，`index` 为变量 `a` 在待检测变量集中的序号。

需要说明的是，由于引用变量未初始化的验证只存在于局部变量，所以本算法中对于待检测变量集合的计数是按照函数声明块进行的，每处理完一个函数定义，待检测变量集合的计数即 `array_k` 数组的大小归零，清空待检测变量集合进入下一个函数声明块的处理。

在 Clang 中，声明语句对应的抽象语法树节点为 DeclStmt 类型节点，声明语句中的每一个声明作为 DeclStmt 的子节点存在，声明引用表达式对应的抽象语法树节点为 DeclRefExpr 类型节点，类型转换表达式对应的抽象语法树节点为 CastExpr 类型节点，一元操作符对应的抽象语

```
//代码变换规则 A：变量值使用 a ->
*(check_uninit(array_k, index, file_path, line_no), &(a)))
//代码变换规则 B：变量赋值 a ->
*(set_uninit(array_k, index), &(a)))
//相关函数的实现
inline void set_uninit(_Bool status[], unsigned index) {
    status[index] = 1;
}
inline void check_uninit(_Bool status[], unsigned index, char *file_path, unsigned line_no) {
    if(status[index] != 1)
        printf("variable uninitialized file:%s line:%d\n", file_path, line_no);
}
```

图 5.10 变量赋值及变量引用代码转换规则

法树节点为 UnaryOperator 类型节点，复合赋值二元操作符对应的抽象语法树节点为 CompoundAssignOperator 类型节点，通过重写 RecursiveASTVisitor 的对应方法可以实现对相应语法树节点的访问。

5.3 本章小结

本章主要介绍了 C 语言的广泛适用性，以及由于其编程高自由度所带来的容易引入缺陷的问题。解释了除 0 错误，整数溢出错误，变量未初始化错误的定义并着重介绍了整数溢出验证相关方面的研究。

然后，分别针对除 0 错误，整数溢出错误，变量未初始化错误，在分析错误产生原因的基础上，利用源代码插桩思想和 Clang 编译器前端，设计并实现了插桩规则，使错误在软件运行过程中能够得到动态验证，准确定位错误的产生位置。

第六章 实验与分析

针对第四章和第五章设计并实现的 MOVEC 语言编译器和常见软件错误验证插桩方法,本章选择了 MiBench 嵌入式 C 语言测试集作为目标系统,用于验证 MOVEC 语言编译器的可用性,同时将其与 AspectC, AspectC++ 进行时间,空间效率的比较,最后验证软件常见错误验证插桩逻辑的正确性。实验过程中记录相关数据,分析实验结果并得出相关结论。

6.1 实验目标系统介绍

使用基准检测测试集对程序进行测试,是评价系统性能重要标准。目前用于特定领域计算能力评价的基准检测测试集包括 Dhrystone^[60], LINPACK^[61], Whetstone^[62], CPU2, MediaBench^[63], 其中 Dhrystone 主要用于整数计算能力评价, LINPACK 主要用于向量计算能力评价, Whetstone 和 CPU2 主要用于浮点计算能力评价, MediaBench 主要用于多媒体解析能力评价。而通用领域的计算能力评价,主要使用 SPEC CPU^[64]测试集和 Mibench 测试集,二者的不同之处在于 SPEC CPU 的测试主要针对服务器,桌面操作系统级的应用,而 Mibench 主要针对嵌入式平台的应用, Mibench 测试集从指令分布,内存行为,并行化等方面都有其独特的优点。C 语言是在嵌入式平台广泛使用的语言,而且随着嵌入式应用需求的不断增加,对于 C 语言程序的验证将更多地集中在嵌入式应用领域,因此,本文实验中选择 Mibench 作为基准检测测试集。

Mibench 测试集包含了 35 个用于进行基准检测的嵌入式应用,以更好地评价一个系统在通用领域的计算能力。根据 EEMBC 的模型,这些应用可以分为六大类,对应不同的嵌入式应用范围,这六大类分别为自动化工业控制类,消费设备类,办公自动化类,网络应用类,安全控制类和通信类。

对于测试集中的每个应用,其主要包含三个部分,一是程序的源代码,二是用于程序编译的 confiure 配置文件和 makefile 编译文件,三是用于程序运行测试的测试输入文件和测试运行 bash 脚本,测试输入文件和测试脚本均分为大数据量和小数据量两个版本。

本文从 Mibench 测试集中选取了如下 10 个应用进行插桩测试, adpcm, basicmath, bitcount, CRC32, dijkstra, patricia, qsort, sha, stringsearch, susan。adpcm 是一个自适应差分脉冲编码调制程序, basicmath 是一个无硬件加速简单数学运算程序, CRC32 是 32 位循环冗余校验码生成程序, dijkstra 是一个基于邻接矩阵求最短路径的程序, patricia 是前缀树的应用程序, qsort 是利用快排算法进行字符串和数字排序的程序, sha 是根据输入利用安全 hash 算法生成 160 位摘要的程序, stringsearch 是利用大小写无关比较算法在短语中搜索指定单词的程序。

6.2 实验软硬件平台介绍

本文实验使用的硬件平台信息包括如下：双核处理器，英特尔酷睿 2 双核 T5470，主频：1.6 GHz，内存 3GB。

本文实验中所使用的软件平台信息如表 6.1 所示。

表 6.1 测试软件平台信息

操作系统	编程语言	开发平台
Fedora 20 64bit	C++, bash	g++, makefile
Windows 7 64bit	C++, bash	Microsof Visual Studio 2013, Mingw

如 4.2.5 节所述，本文中的工具实现了跨平台的使用，所以开发和实验过程所使用的软件平台也相应的划分为两类。在 Linux 系统 Fedora 20 64bit 中，使用了 C++语言作为编程语言，g++ 作为相应编译工具，makefile 用于控制整个工具框架的文件生成和编译，bash 作为测试实验脚本的编写语言。在 Windows 系统 Windows 7 64bit 中，同样使用 C++语言作为编程语言，这样可以使程序文件不作修改应用于不同的平台，Microsof Visual Studio 2013 作为相应编译工具，makefile 用于控制整个工具框架的文件生成和编译，bash 作为测试实验脚本的编写语言，Mingw32 作为提供例如 find 和 rm 等常用工具的工具包。

6.3 实验结果及分析

本文的实验主要分为两个部分，一是 MOVEC 编译器的对比实验，另一个是常见软件错误插桩验证实验，下面将分别对实验结果进行介绍和分析。

6.3.1 MOVEC 编译器实验

对于 MOVEC 编译器和 ACC，AC++编译器的比较实验，本节将从可用性和实际运行数据两个不同的方面进行对比。

6.3.1.1 编译器可用性比较

为了更加准确评价 MOVEC 编译器的性能，我们选取了同样针对 C 的面向方面语言工具 ACC 和 AC++作为对比对象，从编译消耗时间，编译后代码时间效率，编译后代码空间占用三个方面对编译器的实际编译效果进行比较。为了统一评价标准，以下对比实验数据均来自于 Linux 平台实验。

在 Linux 系统下，使用 bash 进行测试脚本的编写，测试脚本主要分为三个部分，原始代码的编译执行，原始代码的插桩，插桩后代码的编译执行。

对于工具 ACC 来说，需要对目标测试应用进行如下修改：1、因为 ACC 并未提供基于项

目文件夹形式的编译，所以需要通过 `bash` 脚本手动搜索应用中的所有 C 文件。2、由于 ACC 只接受经过系统编译器预处理的文件，所以需要在 `bash` 脚本中逐一调用 Gcc 编译器对项目文件和方面文件进行预处理。3、ACC 插桩后的代码不可以直接调用原 `makefile` 编译运行，`makefile` 中所有的链接部分代码需要增加 `spec.o` 表示链接方面代码的目标文件，增加 `-lacc` 表示链接 ACC 专用的函数库。4、ACC 未提供输出指定文件夹功能，实验中采用 `bash` 脚本进行文件夹拷贝。

对于工具 AC++来说，需要对目标测试应用进行如下修改：1、因为 AC++是对 C++程序的面向方面支持工具，虽然 C++语言部分兼容 C 语言，但是 C 语言中依然存在部分编码格式无法被 g++编译工具识别，所以项目中的代码需要修改符合 C++规范。2、因为 AC++编译得到的辅助代码使用了 `template` 等 C++特有语法，所以项目中原有的 `makefile` 需要修改以使用 g++进行编译。3、AC++工具未提供整体输出项目到指定文件夹功能，实验中采用 `bash` 脚本进行文件夹拷贝。

对于工具 MOVEC 来说，其完全兼容 C 语言，支持文件夹形式的编译，支持输出到指定文件夹，插桩后的代码，可以直接调用原有项目 `makefile` 进行编译运行，而不需要作任何修改，因此在工具可用性方面，MOVEC 是优于 ACC 和 AC++。

6.3.1.2 编译器运行比较及分析

对于面向方面编译器的编译，除了项目文件之外，还需要一个相应的方面文件，以定义插入的横切关注点代码。考虑到方面编译器中最常用的是函数相关的接入点，同时为了保证测试覆盖尽可能多的切入点和通知形式，对于每一个项目，其相应方面文件中涉及两个不同的函数切入点，即函数调用切入点和函数执行切入点，并在切入点的基础上构造 `before`、`around` 和 `after` 三种通知，`around` 通知的目标动作中增加对原函数的执行调用，这样每个方面文件就包含了六个不同通知定义。

为了进行插桩代码正确性的比较，上述六个通知的目标执行代码将利用一个变量进行执行次数的记录，在此基础上，增加两个通知定义，分别在 `main` 函数执行后和 `exit` 函数调用前打印变量的统计结果，用以验证通知函数目标动作调用次数。

在利用 `bash` 脚本分别调用 ACC、AC++和 MOVEC 对项目文件夹进行编译后，统计时间和正确性数据如表 6.2 所示，其中 `origin run` 是指未插桩前软件执行指定用例所需要的时间，`MOVEC weave` 是指调用 MOVEC 进行方面编译所需的时间，`MOVEC run` 是指插桩后代码执行指定用例所需要的时间，`MOVEC R` 是指插桩后代码在统计方面动作调用次数时的正确性，对于 ACC 列和 AC++列数据具有类似的意义，数据存在毫秒级误差。

综合表中数据分析可以看出，首先 ACC 和 AC++存在无法正常插桩的情况，`adpcm` 应用对于 ACC 编译出现错误是体现为插桩后代码出现类型不匹配无法正常编译，对于 AC++编译错误

体现为插桩后代码运行出现段错误。

对于方面编译时间即所需插桩时间，MOVEC 远小于 ACC 和 AC++，从平均值数据分析可以发现，ACC 编译的平均时间为 MOVEC 平均时间的 2.8 倍，AC++编译的平均时间为 MOVEC 平均时间的 11.55 倍。

在插桩之后程序运行时间方面，考虑到测试存在毫秒级误差，从平均时间来看，MOVEC
表 6.2 实验 ACC、AC++、MOVEC 插桩时间，正确性及目标程序效率变化

	origin	MOVEC		ACC			AC++			
	run	weave	run	R	weave	run	R	weave	run	R
adpcm	0.05	0.493	0.052	T	#	#	#	#	#	#
basicmath	0.027	0.421	0.030	T	1.010	0.028	T	4.046	0.027	T
bitcount	0.015	0.464	0.016	T	1.405	0.016	F	6.418	0.015	T
crc32	0.025	0.062	0.026	T	0.194	0.027	F	0.762	0.027	T
dijkstra	0.017	0.108	0.018	T	0.280	0.019	T	1.461	0.018	T
patricia	0.037	0.177	0.037	T	0.708	0.036	T	1.569	0.035	T
qsort	0.015	0.157	0.016	T	0.414	0.015	F	1.549	0.014	T
sha	0.009	0.148	0.008	T	0.333	0.008	T	1.488	0.010	T
string	0.003	0.180	0.004	T	0.449	0.003	T	3.577	0.003	T
search										
susans	0.014	0.178	0.015	T	0.508	0.014	T	0.958	0.013	T
ΣSum	0.162	1.895	0.17	#	5.301	0.166	#	21.828	0.162	#
(no adpcm)										
Avg	0.018	0.210	0.018	#	0.589	0.018	#	2.425	0.018	#
(no adpcm)										

和 ACC，AC++基本相同，且和插桩前运行时间相比，不会产生过多时间消耗。

然后，从计数结果方面，ACC 存在计数错误，体现为当多个 around 通知和 after 通知同时存在情况下，ACC 会出现计数过多的情况。

在上述编译实验的基础上，可以获得项目在编译前和编译后代码总量的变化情况如表 6.3 所示，其中 Origin 表示进行方面编译前的代码行总数，Instrumented MOVEC 是指经过 MOVEC 方面编译后的代码行总数，其余列具有类似的含义。

最后，从代码膨胀方面，MOVEC 的平均代码膨胀率为 49%，ACC 的平均代码膨胀率为 596%，AC++的平均代码膨胀率为 199%，所以相比较而言，MOVEC 保持了原有的程序文件结构，对原有代码改动最小。

具体分析实验结果产生原因如下：

对于 ACC 工具无法正常插桩，主要原因是由于面向方面工具所调用的 C 语言解析器的限制，ACC 所调用的 C 语言解析器是利用 GENTLE 编译器构造系统改造而成的，其在识别 AST 树节点类型以及特殊情况处理方面存在不足，例如函数数组参数自动退化为指针。

对于 ACC 和 AC++工具插桩时间远高于 MOVEC，主要原因有以下几个方面，一是 MOVEC 使用的 C 语言解析器 Clang 的效率高于 ACC 使用的 GENTLE 和 AC++使用的 Puma；二是 ACC 和 AC++的预处理过程使头文件全部整合到了源文件当中，带来大量的文件 IO，而 MOVEC 通过改进预处理过程，避免了头文件整合，提高了效率；三是 AC++多了一步知识库的创建和搜索匹配过程，而 MOVEC 是直接在抽象语法树基础上进行搜索匹配，提高了效率。

对于 ACC 和 AC++工具代码膨胀远高于 MOVEC，主要原因是 ACC 和 AC++使头文件全部整合到了源文件当中，而 MOVEC 通过改进预处理过程，避免了头文件整合，降低了插桩后代码的体积，尽可能减少对源代码的改动。

表 6.3 实验 ACC、AC++、MOVEC 插桩后代码膨胀率

	Origin	Instrumented		
		MOVEC	ACC	AC++
adpcm	361	805	0	0
basicmath	401	1409	12035	4331
bitcount	644	756	12489	4760
crc32	187	335	2112	824
dijkstra	352	748	3654	1447
patricia	542	1166	5118	1674
qsort	100	446	4190	1132
sha	241	475	3599	1349
stringsearch	3199	3471	4157	4781
susan	2079	2704	6517	2856
Σ Sum(no adpcm)	7745	11510	53871	23154
Avg(no adpcm)	860	1278	5985	2572

对于 ACC 工具计数出现错误，主要原因是由于 ACC 的插桩逻辑没有正确的处理在插桩过程中接入点的动态变化。如 4.2.4.8 所述，当多个通知同时作用于一个接入点，且存在包含 proceed 调用的 around 通知时，接入点的位置会随着通知的处理进行动态的变化，每处理完一个包含 proceed 调用的 around 通知，当前接入点就应变化为 proceed 调用所在位置，因为 proceed 调用才是当前程序中对原接入点动作的执行。AspectC++，AspectJ 的插桩逻辑中均体现了接入点的动态变化，在 ACC 的插桩逻辑中，接入点是保持不变的，所有的通知都是基于插桩前程序中的接入点进行处理，这并不能准确反映面向方面中 around 通知的真正含义，本文中设计的工具 MOVEC 实现了接入点的动态处理，具体实现逻辑在 4.2.4.8 进行了详细的描述。

综合上述, MOVEC 在时间, 正确性和代码膨胀三个方面优于 ACC 和 AC++工具, 是有效的 AOP 工具。

6.3.2 常见软件错误实验

本文中采用源代码转换的方法实现了对于除 0 错误, 整数溢出错误, 变量使用前未初始化的验证, 为了更加准确考察其正确性, 我们分别对测试程序集中的每一个应用进行验证, 同时为了降低输入输出对程序的影响, 屏蔽了错误验证中的打印语句。实验从插桩消耗时间, 插桩后代码执行效率, 插桩后代码执行结果三个方面统计数据如表 6.4, 其中 origin run 是指未插桩前软件执行指定用例所需要的时间, 除 0 weave 是指调用 MOVEC 进行除 0 插桩所需的时间, 除 0 run 是指插桩后代码执行指定用例所需要的时间, 除 0 R 是指插桩后代码在用例执行时的正确性, 对于整数溢出列和变量使用前未初始化列数据具有类似的意义, 数据可能存在毫秒级误差。

表 6.4 常见错误验证时间和正确性数据

	origin	除 0	整数溢出				变量使用前未初始化			
	run	weave	run	R	weave	run	R	weave	run	R
adpcm	0.052	0.490	0.05	T	0.496	0.302	T	0.503	0.051	T
basicmath	0.027	0.399	0.028	T	0.400	0.033	T	0.411	0.029	T
bitcount	0.018	0.449	0.015	T	0.461	0.364	T	0.461	0.017	T
crc32	0.029	0.066	0.027	T	0.062	0.027	T	0.063	0.025	T
dijkstra	0.018	0.501	0.02	T	0.103	0.09	T	0.104	0.018	T
patricia	0.035	0.175	0.037	T	0.168	0.043	T	0.172	0.037	T
qsort	0.016	0.151	0.016	T	0.152	0.014	T	0.151	0.016	T
sha	0.008	0.144	0.007	T	0.149	0.204	T	0.148	0.009	T
string	0.004	0.185	0.005	T	0.182	0.009	T	0.183	0.004	T
search										
susans	0.012	0.154	0.014	T	0.230	0.372	T	0.214	0.013	T

综合表中数据分析, 经过验证算法插桩后的程序的执行结果没有发生改变, 说明三种验证算法的代码转换逻辑对于程序原有的执行流程不会产生影响, 保证了程序执行的正确性。除 0 错误的验证对程序执行效率的影响很小, 而整数溢出错误的验证由于涉及到运算类型较多, 带来更多的插桩检查, 而且整数溢出错误验证需要进行类型扩展后二次运算, 因此在包含大量整数计算的程序中, 例如 adpcm, bitcount, dijkstra 等, 会使程序运行效率产生较大的影响, 变量使用前未初始化验证对程序执行效率的几乎不产生影响。

下面通过 basicmath 中的一段程序代码, 说明整数溢出验证插桩逻辑在实际应用中的处理

情况，插桩前代码为 $e = (a \ll 1) + 1$ ，插桩后代码如图 6.1 所示。

通过分析可以看出，在代码 $e = (a \ll 1) + 1$ 中涉及了三次溢出验证过程，分别对应加法操作，右移操作和常数 1 的隐含类型转换操作。从代码逻辑上，加法和隐含类型转换的验证过程，采用了类型提升并进行运算比较的验证策略，而右移操作采用了移位长度与位长进行比较的验证策略。从代码设计上，利用函数调用中参数计算的特殊运算顺序，保证了验证过程各个运算顺序的正确性；利用临时变量和数据栈，保证了验证过程各个运算次数正确性，避免了副作用。从插桩过程上，通过抽象语法树的后序遍历，保证三次验证过程代码的插桩不会出现交叉，符合 C 语言的语法。

```
e = ((unsigned long)over_bo(
    (unsigned long)overflow_temp_1 + (unsigned long)overflow_temp_2,
    overflow_temp_1 + overflow_temp_2,
    "/home/david/out/isqrt.c", 57,
    (overflow_temp_1 = ((a << ((unsigned long)over_sh(overflow_temp_1, sizeof(unsigned long),
"/home/david/ out/isqrt.c", 57, overflow_temp_1 = (1))))),
    overflow_stack_push(overflow_temp_1),
    overflow_temp_2 = (((unsigned long)over_bo((unsigned long)overflow_temp_1, overflow_temp_1,
"/home/david/out/isqrt.c", 57, overflow_temp_1 = (1))))),
    overflow_temp_1 = overflow_stack_pop(),
    0));
```

图 6.1 验证插桩代码实例

从实验中可以看出，利用 Clang 编译器框架和本文中的源代码插桩的算法，对程序执行除 0 错误，整数溢出和变量使用前未初始化验证是一种可行的方案。

6.4 本章小结

本章首先利用 ACC、AC++、MOVEC 在 Mibench 测试集上进行相同方面文件条件下编译性能的测试，通过实验的时间数据和代码行数据，证明了 MOVEC 可以在保证对原程序性能的低影响前提下，实现更低的插桩时间和更小的代码膨胀率，对源代码的改动更小，有利于二次阅读和开发。

然后分别在 Mibench 测试集上进行三种错误验证的测试，通过程序运行结果的比较，验证了其中源代码转换规则的正确性，通过时间数据考察了插桩对目标程序性能的影响。

第七章 总结与展望

7.1 论文总结

随着软件系统在社会生活中的广泛应用,对软件可靠性的要求也越来越高。目前软件验证的技术主要包括静态分析,模型检测,软件测试,但这些验证方法均不同程度上无法反映系统实际运行环境中的状态,而且部分存在复杂度较高的问题。运行时验证作为一种轻量级的验证方法,既具备了系统实时监控的能力,又综合了模型检测的性质描述方法。将监控器和验证器集成至目标系统中,是实现运行时验证的一个技术关键,但是,由于软件规模和复杂度的不断增加,手动插桩几乎是不现实的。随着编译器技术的不断发展,为实现监控和验证代码插桩自动化提供了支持,同时面向方面编程概念的提出,使监控代码插桩需求的模块化描述和管理成为了可能。本文设计并实现了针对 C 的面向方面语言 MOVEC 及其编译器,以支持监控器插桩位置描述的模块化管理,同时基于 Clang 编译器源代码插桩技术实现了常见错误的验证,最后将其应用在基准测试集,检验编译器相关性能和验证插桩逻辑正确性,得出相关结论。

在这一段时间的研究和实验过程中,主要的研究内容和成果如下:

(1) 研究和分析了编译器的编译流程,包括每一步的输入输出和编译逻辑。进一步研究和分析了 LLVM 编译框架,并着重介绍了其中 Clang 编译器前端的处理流程和 AST 树的数据结构。

(2) 介绍软件工程中和监控器插桩相类似的横切关注点分离技术,并着重解释了应用最广泛的面向方面编程的基本概念和实现技术关键。

(3) 设计并实现了针对 C 程序的面向方面语言 MOVEC,为监控器插桩位置的描述和监控器的模块化管理提供了支持。

(4) 利用 Clang 编译器的代码分析插桩方法,设计并实现 C 语言程序中常见错误(除 0,整数溢出,未初始化变量引用)的检测,实践了 Clang 的源代码转换技术。

(5) 利用基准测试集 Mibench,进行 MOVEC,ACC,AC++三者的准确度,时间消耗和空间占用对比,得出 MOVEC 实用性的相关结论。同时,对常见软件错误验证插桩算法进行正确性验证和性能考察。

7.2 工作展望

本文基于 Clang 编译器前端和源代码插桩技术,参照面向方面编程的概念,设计实现了面向方面语言 MOVEC 作为插桩需求的描述语言,通过实验考察了 MOVEC 语言编译器的性能和实用性,实现了常见软件错误的运行时验证插桩。目前所做的工作还存在许多不足之处,进一步的工作包括以下几个方面:

(1) 针对 C 语言程序中存在的其他语法结构，例如 for 循环语法结构，switch 选择语法结构，引入作为接入点，扩展 MOVEC 语言的插桩位置描述能力。

(2) MOVEC 编译器的项目级编译目前只能在 Linux 平台上进行，考虑到 Windows 平台开发依然占有很大比例，可以在 Windows 平台实现 json 文件的生成，以增强 MOVEC 编译器的实用性。

(3) 研究多核心并行编译技术，将 MOVEC 编译器对文件的解析插桩过程分散到不同的处理器核心上进行，提高 MOVEC 对整体项目编译的时间性能。

(4) 本文中设计的常见软件错误的插桩验证工具，虽然保证了对所有可能产生错误的位置不遗漏检测，但是大量检测代码的插桩会对目标程序的性能产生影响。因此，可以结合静态分析技术，对所有可能产生错误的位置进行一次前置分析，例如当除数为非零常数时无需插桩，尽可能降低插桩点的数量，减少检测所带来的性能开销。

参考文献

- [1] A. Finkelstein, J. Kramer, The future of software engineering, Association for Computing Machinery, 2000.
- [2] L. Bilge, T. Dumitras, Before we knew it: an empirical study of zero-day attacks in the real world, in: Proceedings of the 2012 ACM conference on Computer and communications security, ACM, 2012, pp. 833-844.
- [3] M. Dowson, The Ariane 5 software failure, ACM SIGSOFT Software Engineering Notes, 22 (1997) 84.
- [4] IEEE, Standard Glossary of Software Engineering Terminology, IEEE Software Engineering Standard, (1990) 610.612-190.
- [5] M.R. Lyu, Handbook of software reliability engineering, IEEE computer society press CA, 1996.
- [6] Z. Zhidou, S. Short, Y. Roudier, Static Code Analysis for Software Security Verification: Problems and Approaches, in: Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International, IEEE, 2014, pp. 102-109.
- [7] Z. Chen, G. Motet, Nevertrace claims for model checking, in: Model Checking Software, Springer, 2010, pp. 162-179.
- [8] W.C. Hetzel, B. Hetzel, The complete guide to software testing, John Wiley & Sons, Inc., 1991.
- [9] M. Leucker, C. Schallhart, A brief account of runtime verification, The Journal of Logic and Algebraic Programming, 78 (2009) 293-303.
- [10] C. MacNamee, D. Heffernan, On-Chip Instrumentation for Runtime Verification in Deeply Embedded Processors, in: VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on, IEEE, 2015, pp. 374-379.
- [11] N. Decker, F. Kuhn, D. Thoma, Runtime verification of web services for interconnected medical devices, in: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, IEEE, 2014, pp. 235-244.
- [12] P. Daian, Y. Falcone, P. Meredith, T.F. Şerbănuţă, A. Iwai, G. Rosu, RV-Android: Efficient Parametric Android Runtime Verification, a Brief Tutorial, in: Runtime Verification, Springer, 2015, pp. 342-357.
- [13] S. Fischer, M. Leucker, Runtime verification and reflection for wireless sensor networks, in: Software Engineering for Sensor Network Applications (SESENA), 2013 4th International

- Workshop on, IEEE, 2013, pp. 35-36.
- [14] A. Pnueli, The temporal logic of programs, in: Foundations of Computer Science, 1977., 18th Annual Symposium on, IEEE, 1977, pp. 46-57.
 - [15] V. Stolz, E. Bodden, Temporal assertions using AspectJ, Electronic Notes in Theoretical Computer Science, 144 (2006) 109-124.
 - [16] P. Avgustinov, J. Tibble, E. Bodden, L. Hendren, O. Lhoták, O. De Moor, N. Ongkingco, G. Sittampalam, Efficient trace monitoring, in: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, ACM, 2006, pp. 685-686.
 - [17] C. Colombo, G.J. Pace, G. Schneider, LARVA---safer monitoring of real-time Java programs (tool paper), in: Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on, IEEE, 2009, pp. 33-37.
 - [18] M. Leucker, Teaching runtime verification, in: Runtime Verification, Springer, 2012, pp. 34-48.
 - [19] D. Bartetzko, C. Fischer, M. Möller, H. Wehrheim, Jass—Java with assertions, Electronic Notes in Theoretical Computer Science, 55 (2001) 103-117.
 - [20] F. Chen, G. Roşu, Java-MOP: A monitoring oriented programming environment for Java, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2005, pp. 546-550.
 - [21] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Java-MaC: a run-time assurance tool for Java programs, Electronic Notes in Theoretical Computer Science, 55 (2001) 218-235.
 - [22] K. Havelund, G. Roşu, Monitoring java programs with java pathexplorer, Electronic Notes in Theoretical Computer Science, 55 (2001) 200-217.
 - [23] H. Barringer, K. Havelund, TraceContract: A Scala DSL for trace analysis, Springer, 2011.
 - [24] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2004, pp. 44-57.
 - [25] H. Barringer, D. Rydeheard, K. Havelund, Rule systems for run-time monitoring: from Eagle to RuleR, Journal of Logic and Computation, 20 (2010) 675-706.
 - [26] E. Yourdon, L.L. Constantine, Structured design: Fundamentals of a discipline of computer program and systems design, Prentice-Hall, Inc., 1979.
 - [27] M. Fayad, D.C. Schmidt, Object-oriented application frameworks, Communications of the ACM, 40 (1997) 32-38.
 - [28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, Springer, 1997.

- [29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: ECOOP 2001—Object-Oriented Programming, Springer, 2001, pp. 327-354.
- [30] J. Bonér, AspectWerkz-dynamic AOP for Java, in: Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD), Citeseer, 2004.
- [31] A. JBoss, JBoss AOP-Aspect-Oriented Framework for Java, JBoss Inc.[online] <http://docs.jboss.org/aop/1.1/aspect-framework/reference/en/html/index.html> (accessed 5 March 2012), (2004).
- [32] K. Sirbi, P.J. Kulkarni, Stronger enforcement of security using aop and spring aop, arXiv preprint arXiv:1006.4550, (2010).
- [33] M. Gong, Z. Zhang, H.-A. Jacobsen, AspeCt-oriented C for systems programming with C, in: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), 2007.
- [34] O. Spinczyk, D. Lohmann, M. Urban, Advances in AOP with AspectC++, SoMeT, 129 (2005) 33-53.
- [35] A. Jackson, S. Clarke, Sourceweave. net: Cross-language aspect-oriented programming, in: Generative Programming and Component Engineering, Springer, 2004, pp. 115-135.
- [36] T. Aslam, J. Doherty, A. Dubrau, L. Hendren, AspectMatlab: An aspect-oriented scientific programming language, in: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, ACM, 2010, pp. 181-192.
- [37] 王亚普, 王志坚, 叶枫, 等. 增强云服务可信性的研究. 微电子学与计算机, 2012, 11.
- [38] 徐刚, 宋巍, 胡昊等, 一种支持过程动态更新的过程系统设计与实现. 计算机科学, 2012: 1-19.
- [39] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques, Addison wesley, 1986.
- [40] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE, 2004, pp. 75-86.
- [41] J. Levine, Flex & Bison: Text Processing Tools, " O'Reilly Media, Inc.", 2009.
- [42] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton Jr, N degrees of separation: multi-dimensional separation of concerns, in: Proceedings of the 21st international conference on Software engineering, ACM, 1999, pp. 107-119.
- [43] K.J. Lieberherr, I. Silva-Lepe, C. Xiao, Adaptive object-oriented programming using graph-based customization, Communications of the ACM, 37 (1994) 94-101.
- [44] W. Harrison, H. Ossher, Subject-oriented programming: a critique of pure objects, ACM, 1993.

- [45] G. Kiczales, J. Des Rivieres, D.G. Bobrow, The art of the metaobject protocol, MIT press, 1991.
- [46] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM, 44 (2001) 51-57.
- [47] 唐祖锴, 彭智勇, 面向方面程序设计语言研究综述. 计算机科学与探索, (2010) 1-19.
- [48] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S.A. Smolka, S.D. Stoller, E. Zadok, InterAspect: Aspect-oriented instrumentation with GCC, Formal Methods in System Design, 41 (2012) 295-320.
- [49] R. Douence, M. Südholt, A model and a tool for event-based aspect-oriented programming (EAOP), Techn. Ber., Ecole des Mines de Nantes. TR, 2 (2002) 34-38.
- [50] S. Schonger, E. Pulvermüller, S. Sarstedt, Aspect oriented programming and component weaving: Using XML representations of abstract syntax trees, in: Workshop Aspektorientierte Softwareentwicklung, 2002.
- [51] I. Nagy, R. Engelen, D. Ploeg, An overview of Mirjam and WeaveC, (2007).
- [52] N. Delgado, A.Q. Gates, S. Roach, A taxonomy and catalog of runtime software-fault monitoring tools, Software Engineering, IEEE Transactions on, 30 (2004) 859-872.
- [53] W. Dietz, P. Li, J. Regehr, V. Adve, Understanding integer overflow in C/C++, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 760-770.
- [54] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, D. Song, RICH: Automatically protecting against integer-based vulnerabilities, Department of Electrical and Computing Engineering, (2007) 28.
- [55] D. Brumley, D. Song, J. Slember, Towards automatically eliminating integer-based vulnerabilities, in, Carnegie Mellon University, Tech. Rep. CMU-CS-06-136, 2006.
- [56] P. Chen, Y. Wang, Z. Xin, B. Mao, L. Xie, Brick: A binary tool for run-time detecting and locating integer-based vulnerability, in: Availability, Reliability and Security, 2009. ARES'09. International Conference on, IEEE, 2009, pp. 208-215.
- [57] D. Molnar, X.C. Li, D. Wagner, Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs, in: USENIX Security Symposium, 2009.
- [58] T. Wang, T. Wei, Z. Lin, W. Zou, IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution, in: NDSS, Citeseer, 2009.
- [59] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, The ASTRÉE analyzer, in: Programming Languages and Systems, Springer, 2005, pp. 21-30.
- [60] R.P. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM, 27 (1984) 1013-1030.

- [61] J.J. Dongarra, J.R. Bunch, C.B. Moler, G.W. Stewart, LINPACK users' guide, Siam, 1979.
- [62] H.J. Curnow, B.A. Wichmann, A synthetic benchmark, The Computer Journal, 19 (1976) 43-49.
- [63] C. Lee, M. Potkonjak, W.H. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, 1997, pp. 330-335.
- [64] B. Case, Spec2000 retires spec92, The Microprocessor Report, 9 (1995).

致 谢

本文是对我攻读硕士学位期间研究工作的一个总结。首先，我要衷心感谢李绪蓉老师，虽然李老师没有直接对我的学习研究进行指导，但是当在程序性事务上遇到问题时，总能第一时间得到李老师的帮助，使问题得到很好的解决。其次要感谢在三年的研究生学习生活中带领我前进的陈哲老师，自从我被录取成为研究生起，便悉心为我制定学习目标，帮助我培养独立研究和独立开发的能力，他丰富的学识，严谨的研究态度和平易近人的性格给我以及所有的学生留下了深刻的印象，同时也促使了我在学术研究上向老师看齐，保持严谨认真的治学态度。

感谢南京航空航天大学计算机科学与技术学院的所有老师和同学们，在这三年里，我们作为一个集体，互相帮助，积极参加学校活动，增进了师生间和同学间的友谊，为学习提供了一个良好的氛围。

感谢王哲民，李昕等同学，在研究生的学习生活中，我们在学习上共同探讨解决问题，生活上互相帮助，共同完成了两年半的研究生学习路程。感谢沈宏，陈骁，邱海波三位室友在平时的生活给予的帮助，有了他们，研究生的三年生活变得更加精彩。

感谢我的父母，没有他们的支持，我或许不能坚持到研究生的阶段，也就不会通过研究生的学习掌握独立解决问题的能力，我希望今后我能更好的报答他们，让父母的晚年生活更加幸福快乐。

最后感谢各位参与论文评审和答辩的老师，你们辛苦了，祝你们身体健康、工作顺利！

在学期间的研究成果及发表的学术论文

攻读硕士学位期间发表（录用）论文情况

1. 朱云龙,陈哲,王哲民,李绪蓉,黄志球. 针对 C 语言的面向方面语言设计与实现. 小型微型计算机系统 (已录用)
2. Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, Zhibin Yang. Parametric Runtime Verification of C Programs. In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016), to appear. Springer, 2016

攻读硕士学位期间申请的专利情况

1. 一种源代码中值计算错误的自动检测和定位方法, 申请号: 201410499170.9 公开号: CN104298594A.
2. 复杂约束条件下多组件软件部署的自动化与自适应方法及其部署管理系统, 申请号: 201410500293.X 公开号: CN104298525A

攻读硕士学位期间申请的参加科研项目情况

1. 国家自然科学基金 (61100034): 基于控制机制的软件可靠性新技术及其理论研究
2. 国家自然科学基金委员会-中国民航局民航联合研究基金 (U1533130)
3. 教育部留学回国人员科研启动基金 (2013)
4. 中央高校基本科研业务费专项资金 (NS2016092)

攻读硕士学位期间获奖情况

1. 第十二届全国研究生数学建模竞赛, 三等奖
2. 第四届中国软件杯大学生软件设计大赛, 优秀奖