****



**研 究 生 毕 业 论 文**

**（申请硕士学位）**

**论文题目** 循环不变式生成方法研究

**作者姓名** 刘自恒

**学科、专业名称**  计算机软件与理论

**研 究 方 向**  信息安全

**指导教师** 曾庆凯 教授

二零一二年 五月 二十日

**学 号：MG0933021**

**论文答辩日期： 年 月 日**

**指 导 教 师： （签字）**

**Nanjing University**

(Master Degree Dissertation)

**Research on Static Software Security**

**Flaw Detection**

By

**Zi-heng Liu**

Supervised by

Professor **Qing-kai Zeng**

Department of Computer Science and Technology

Nanjing University

May 2012

Nanjing, P.R.China

# 摘要

软件作为信息系统的实现载体，广泛应用在各个领域，软件中的任何安全漏洞或错误的实现都可能导致非常严重的后果。通过大量的测试可以提高软件的可靠性，但成本较高，也不能完全保证软件的可靠。形式化证明方法基于严密的数学和逻辑基础，经过正确性证明的程序可以保证软件符合指定的程序规范，从而大大提高软件可靠性。

程序正确性证明的一个关键问题是发现充分的循环不变式来辅助、支撑证明过程的完成。由人工寻找循环不变式工作繁琐，容易出错。因此，研究如何自动地有效分析和生成循环不变式，使得软件证明过程顺利完成，成为形式化方法研究领域的一个重要研究课题。本文主要针对软件验证技术中的循环不变式生成技术进行研究和实现：

（1）对形式化方法主要理论进行总结。本文根据形式化用到的主要理论的不同对形式化技术进行总结。对各自的理论核心、主要解决的问题、典型工具进行了阐述。在此基础上，指出形式化证明过程的难点。

（2）介绍和总结了目前求解循环不变式的主要方法。根据他们解决问题范围的不同进行阐述，指出了目前工作存在的不足。

（3）本文提出了一种基于条件赋值转换和自适应模板技术的循环不变式生成方法。根据程序控制流结构、依赖分析结果和值分析结果，生成可能的循环不变式，然后由定理证明器求解。该方法可以自动地运行，并将结果以注释的形式添加到程序中，辅助完成程序属性的证明过程。

（4）基于上述方法，在Frama-C平台和APRON库的基础上，实现了一个分析插件。选取了一些程序作为分析对象，与其他方法作了比较，结果表明可以发现更多的不变式，使一些程序规范顺利得到证明。

**关键字：验证、程序规范、循环不变式、自动化、插件**

# Abstract

Software has been used widely in various areas as the carrier of information systems and any vulnerability or incorrect development in software will cause highly serious failures. It can increase the confidence with enough testing which will be very expensive. But unfortunately it still cannot guarantee that the software is fully reliability. Formal methods are based on strict mathematically and logically basis. It can ensure that the software meets its given specifications through proving of correctness, which greatly improves the reliability of software.

A key issue in proving is finding sufficient loop invariants to assist the automatic program proving. However, writing loop invariants manually is boring and error-prone. Thus, the research on how to analysis and generate loop invariants automatically becomes an important issue in the area of formal methods. This dissertation focused on the generation of loop invariants and mainly covers the subjects of theory of formal methods and generation technologies for invariants.

(1)We summarized the main theories of formal methods. It is focused on the core theory, the main problem to resolve and classical tools. Then we point out the difficulty of formal proof.

(2) We summarized the methods for generating loop invariants, including the kind of program to be analyzed, the way how they work and the shortcomings.

(3) In this thesis, we propose an approach based on conditional assignment and adaptive template. It generates potential loop invariants according to CFG, PDG and value analysis which are then resolved by theory prover. It can run automatically and insert invariants as ACSL comments.

(4)Based on the method above, we designed and implemented an plugin named loopInv with Frama-C and APRON. According to the experiments on some programs, it can generate more invariants comparing with other tools.

**Keywords：verify, program specification, loop invariant, value analysis, plugin**

# 目录

[摘要 I](#_Toc318666036)

[Abstract II](#_Toc318666037)

[目录 III](#_Toc318666038)

[图片目录 IV](#_Toc318666039)

[表格目录 IV](#_Toc318666040)

[第一章 引言 1](#_Toc318666041)

[1.1 研究背景 1](#_Toc318666042)

[1.2 研究内容 2](#_Toc318666043)

[1.3 本文组织 3](#_Toc318666044)

[第二章 形式化方法基础 4](#_Toc318666045)

[2.1 基本理论 4](#_Toc318666046)

[2.2 主要方法 7](#_Toc318666047)

[2.2.1 模型检测 7](#_Toc318666048)

[2.2.2 定理证明 8](#_Toc318666049)

[2.3 难点 8](#_Toc318666050)

[第三章 循环不变式生成技术研究 10](#_Toc318666051)

[第四章 循环不变式生成技术改进 12](#_Toc318666052)

[第五章 loopInv插件的设计与实现 13](#_Toc318666053)

[第六章 实验过程与结果 15](#_Toc318666054)

[第七章 结论与展望 16](#_Toc318666055)

[参考文献 17](#_Toc318666056)

[攻读硕士学位期间参与的科研项目 19](#_Toc318666057)

[攻读硕士学位期间完成的论文 19](#_Toc318666058)

[致谢 20](#_Toc318666059)

# 图片目录

[**图 1.1 CVE 数据库近10年来的脆弱性数量统计** 1](#_Toc317862373)

[**图 5.2 Frama-C结构图** 12](#_Toc317862374)

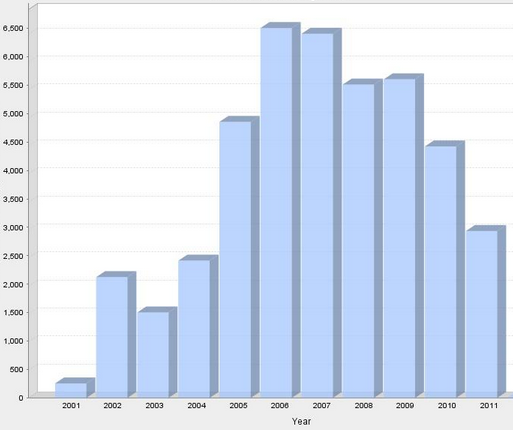
# 表格目录

[**表 6.1 比较** 13](#_Toc317860539)

# 第一章 引言

## 研究背景

随着计算机科学的迅速发展和信息化时代的深刻变革，信息系统已经深入到人类生产生活的各个领域，软件作为信息系统的实现载体，广泛应用在各个领域，软件的任何漏洞都可能导致非常严重的后果。无论是软件设计人员还是软件的用户，均希望在软件系统投入正式使用之前，能得到软件正确性的保证，避免造成软件后期维护成本的上升和使用有缺陷软件造成的损失。软件的安全性一直是系统安全性、可靠性、稳定性的一个重要方面。已经有很多系统由于软件缺陷的存在，造成了巨大的损失，CVE（[cve.mitre.org](http://cve.mitre.org/)）不断发布最新的软件安全漏洞报告，由图 1.1（[nvd.nist.gov](http://nvd.nist.gov/)对CVE数据的统计）可以看出，软件的脆弱性形势一直十分严峻，软件的安全性和可靠性一直困扰着软件开发和使用人员。



**图 1.1 CVE 数据库近10年来的脆弱性数量统计**

另一方面，软件的实现是否和设计需求和规范是否一致决定了软件的正确性，不正确的软件开发实现同样会带来重大损失。

首先用来试图得到无错误软件的方法是测试；通过在程序上运行一组输入集合并检查软件的行为是否符合预期；对于关键性软件，必须达到很高的测试覆盖度才可以获得相对较高的可信度，但是这样的测试输入有时候甚至是无穷的，因而代价很高[[Mauborgne 2004](#_ENREF_35)]。

为了最大限度地保证软件的正确性和无脆弱性，从另一个角度考虑，如果可以从理论上证明程序符合预期的行为或者没有某种错误，同样可以认为软件是可靠的。这需要开发人员对已编码软件的正确性进行分析证明，同时证明软件中没有某些导致软件安全缺陷的代码实现。正确性是指代码的实现和设计的算法目标一致，代码的执行与算法的预期是相同的，代码没有错误地实现算法。没有安全缺陷是指代码的实现过程没有引入可能导致软件崩溃、被其他恶意软件利用篡改等的部分。

证明的过程需要对算法、编码和安全漏洞有着深刻的认识，需要严密的逻辑演算过程，同时也会有一些琐碎的证明过程。这样的过程如果全部由人工来完成，无疑需要大量的人力、物力、财力，而且人工实施难免会出差错。

最早从50年代开始出现了形式化方法（formal methods）的程序设计思想，即J.Backus提出BNF描述Algol60语言的语法，以严格的数学和逻辑理论为基础，适合于软件和硬件系统的描述、开发和验证。经过多年的研究，取得了大量、重要的成果，逐渐融入软件开发周期的各个阶段。在编码后期的检查阶段，使用形式化方法可以自动地帮助分析人员发现其它方法不容易发现的系统描述不一致性或脆弱性，增加软件开发人员对系统的理解，实现开发可靠的软件产品的目标。形式化方法并不排斥测试，但可以认为是一种更好的测试框架[[Airchinnigh 1995](#_ENREF_1)]。形式方法的主要研究内容包括:形式规范说明 (formal specification)和形式验证(formal verification)。基于模型的；基于公理系统和过程代数的

## 研究内容

本文主要包括研究内容如下

（1）阐述了形式化方法的主要理论基础，总结了现有的形式化方法的主要种类。

主要从前置条件、后置条件、不变式、状态机等方面阐述，比较了常用的形式化方法的特点、适用范围、典型工具、局限等内容。

（2）分析、总结、比较了现有的生成循环不变式的方法。

本文列出了目前使用的方法的针对情形，指出了用到的关键算法，指出了各自的特点和不足，并提出了可能的解决方法。

（3）针对数值型计算程序和含有数组的程序，基于条件赋值转换和自适应模板生成技术，提出了改进的循环不变式生成方法。

（4）基于Frama-C和APRON库，将上述方法实现为Frama-C的一个插件loopInv。使用该插件可以自动地完成循环不变式的生成和插桩工作，辅助后续的证明过程。

（5）选取若干样例程序，与其他不变式生成工具做对比。

选取了若干简单示例程序和实际部署使用的程序，计算其中存在的不变式，与其他工具对比，并人工统计分析生成的不变式的质量。

## 本文组织

本文的组织结构如下：

第二章阐述了形式化方法的主要理论基础，给出了目前常用的形式化方法，

第三章探讨生成循环不变式的方法，给出了目前常用的形式化方法的理论核心、主要解决的问题、典型工具。

第四章提出了一种自动生成循环不变式的方法，该方法基于条件赋值转换和自适应模板生成技术，自动化程度高，生成尽可能多的不变式。

第五章基于Frama-C和APRON库，将上述方法加以实现，成为Frama-C的一个插件。

简要介绍了Frama-C的主要功能和特点，介绍了APRON库的作用和效果。遵从Frama-C的开发要求，设计了一个Frama-C的插件。描述了该插件的主要组成、工作原理和实现的主要算法。

第六章选取了一些样例程序，将上述方法和现有方法的效果加以对比；选取了一些实际中使用的程序，对生成的结果进行分析，评价生成的不变式的质量。

第七章总结了目前的工作，并对下一步的研究方向进行了展望。

# 第二章 形式化方法基础

软件开发自动化技术是提高软件生产率的根本途径之一。软件自动化的前提是形式化，形式化不仅仅是对用户需求，而且也是对整个软件系统的严格定义。传统的软件开发方法由于大量的使用自然语言和多种图形符号,结果是尽管经历了仔细地复审，最后的系统规约说明中仍然包含歧义的、含糊的、矛盾的、不完整的需求描述及混乱的抽象层次。使用形式化方法可以克服这些缺点，有理论优势。本章首先以形式化的方式定义相关概念和操作，然后阐述形式化分析方法的理论，阐述理论核心、主要解决的问题和典型工具。

## 基本理论

本文研究的程序使用C语言编写，由一系列指令组成，是人工可读的。在分析中，程序P使用抽象语法树（Abstract Syntax Tree，AST）[[V.Aho, S.Lam et al. 2007](#_ENREF_46)]表示，它是[源代码](http://zh.wikipedia.org/wiki/%E6%BA%90%E4%BB%A3%E7%A0%81)的抽象[语法](http://zh.wikipedia.org/wiki/%E8%AF%AD%E6%B3%95%E5%AD%A6)结构的[树](http://zh.wikipedia.org/wiki/%E6%A0%91_(%E5%9B%BE%E8%AE%BA))状表现形式。

**定义2.1（控制流图）**一个程序（函数）的控制流图（Control Flow Graph, CFG）可定义为

其中，是程序控制点集合，是程序变量集合，是边的集合，是初始位置，是终止位置，每条边由一个转换关系标记，是程序状态的集合，是程序变量的当前状态，是程序变量的下一状态，代表程序变量的初始值。所谓状态，是指程序中变量在当前控制点的取值情况和变量之间的约束关系情况。

程序从CFG的初始位置以初始值开始执行，到达终止位置时结束。程序的控制流图可以从其抽象语法树计算得到。是一个简单的程序，由G. Necula 等人开发的CIL工具[[Necula, McPeak et al. 2002](#_ENREF_40); [Necula, McPeak et al. 2009](#_ENREF_39)]生成的CFG表示如所示。

void phase()

{

int x=0,y=10;

while(x<=y)

{

x++;

}

}

**定义2.2（转换系统）**[[Manna and Pnueli 1995](#_ENREF_34)]提出可以将程序以一阶逻辑形式建模为一个转换系统。转换系统由三部分组成[[Bradley and Manna 2008](#_ENREF_7)]

* 变量集合
* 初始条件
* 指定转换关系的转换公式

其中，表示变量下一状态的取值，函数表示了CFG中每条边的具体语义信息。可以对转换系统添加一些限制，得到不同的结果，如限制变量只能是布尔值，限制转换关系是线性转换、多项式转换等。转换系统的生成可以通过对CFG的遍历得到，在遍历的过程中对每条边赋予具体的语义信息。

迭代计算其中的不动点。

程序规范

**定义2.3（Hoare三元组）**Hoare三元组是具有如下形式的式子，



它表示当条件P成立时，执行语句stmt后Q就成立。

**定义2.3（最弱前置条件）**保证断言有效，为安全属性、终止生成最通用反例



称使得式子总是成立的函数为最弱前置条件计算函数。

例子



这里的最弱前置条件计算方法是通过将属性中的用替换得到的。

属性的验证：对于函数f，假定我们有前置条件P和后置条件Q，证明f是否满足其规范即证明是否成立。如果我们有最弱前置条件函数，由定义有成立。如果可以证明，则可以使用最弱前置条件这一中间结果来证明函数规范，即是有如下推导关系



这是通过计算最弱前置条件证明程序正确性的核心思想：从Q出发，计算经过所有语句直到f起始点的最弱前置条件W，将提交给定理证明器，如果这个关系（称为证明责任，proof obligation）成立则就可以导出Q是有效的。

一些计算最弱前置条件的规则如下。

最强后置条件：得到程序的可达状态集，当没有断言时不能保证精度。

**定义2.4（有限状态机）**一个有限状态机（Finite State Machine，FSM）可记为



其中，是输入字母表，是状态的非空集合，是初始状态，是状态转移函数，是最终状态集合，即状态机接受的状态集合，它可以为空。用FSM可以建模表示程序，以状态机状态的变化来表示程序执行状态的转换；同时可以表示程序属性，它特别适合表示具有时序关系的程序属性。通过求解状态机接受的输出来判断程序是否满足或违反特定的性质。这是模型检测方法的理论基础核心。

循环结构是程序设计的三大结构（顺序结构，分支结构，循环结构）之一，广泛存在于各种程序之中。

**定义2.5（不变式）**如果对于每个计算，则式子是转换系统的一个不变式。计算是指从起始状态开始的一系列程序状态序列。

**定义2.6（归纳不变式）**满足下面两个条件的称为归纳不变式

* 初始时成立：
* 执行转换时一直保持成立：

定义2.7（不动点）

抽象解释[[Cousot and Cousot 1977](#_ENREF_20); [Cousot and Cousot 1979](#_ENREF_21)]是一种抽象语义分析理论，它将程序执行时的变量值映射到抽象域中，在抽象域中定义了一个抽象格，程序的执行语义被抽象为格运算，程序的执行过程转换为抽象域中的运算过程，运算的结果又具体化为程序语义信息。为了使得这个计算过程可终止即收敛，会采取一些近似的技术来保证或加快收敛。抽象格定义为，其中

* 是格中元素的集合
* 是全序关系
* 是格中的最小元素
* 是格中的最大元素
* 是元素间的并操作
* 是元素间的交操作

为了将具体程序和抽象域连接起来，定义抽象函数将具体的程序状态映射为中的元素，定义具体化函数将抽象元素映射为具体的程序状态。抽象元素总是过近似地表示程序行为，即满足如下性质







过去的30多年中，人们研究提出了多种数值抽象域模型并进行了实现[[Cousot and Halbwachs 1978](#_ENREF_23); [Simon, King et al. 2003](#_ENREF_45); [Clariso and Cortadella 2004](#_ENREF_14); [Miné 2006](#_ENREF_36); [Chen, Mine et al. 2008](#_ENREF_11); [Jeannet and Miné 2009](#_ENREF_33)]，几种常见的数值抽象域模型形象表示如。根据在抽象域中的结果来静态地分析判断程序的执行状态。

依赖分析

形式化的程序分析主要由程序规范说明、程序和属性建模、模型分析过程几个阶段组成。

## 主要方法

由于形式化方法具有的优点，该类型方法已经得到了广泛的应用。根据不同的理论和分析目标，出现了多种比较成熟的技术，并取得了很好的效果。形式化的方法已经融入到软件开发的生命周期中。下面介绍几种常见的形式化技术。

### 模型检测

模型检测（model checking）[[Clarke, Grumberg et al. 2000](#_ENREF_17)]是一种基于自动机理论的自动验证技术，最早由Clarke和Emerson[[Clarke and Emerson 1981](#_ENREF_16)]以及Quielle和Sifakis[[Queille and Sifakis 1982](#_ENREF_41)]分别提出的，主要通过搜索状态空间或不动点计算来验证有限状态空间的性质。它可以自动执行，并在程序不满足属性的时候自动构造给出反例路径供开发人员分析，因此取得了令人瞩目的成功，具有很高的使用价值，出现了一些典型的工具，如MOPS[[Chen and Wagner 2002](#_ENREF_10); [Chen, Dean et al. 2004](#_ENREF_9)]、SLAM[[Ball and Rajamani 2002](#_ENREF_3); [Ball, Cook et al. 2004](#_ENREF_2)]、BLAST[[Henzinger, Jhala et al. 2003](#_ENREF_31); [Beyer, Henzinger et al. 2007](#_ENREF_5)]、MAGIC[[Chaki, Clarke et al. 2003](#_ENREF_8)]、SPIN[[Holzmann 1997](#_ENREF_32)]、JPF[[Havelund and Pressburger 2000](#_ENREF_29)]、NuSMV[[Cimatti, Clarke et al. 1999](#_ENREF_13); [Cimatti, Clarke et al. 2002](#_ENREF_12)]等，并辐射到了工业界。

模型检测存在的主要问题是状态空间爆炸。由于很多程序的状态理论上是无穷的，同时内存和CPU等物理资源是有限的，因而当程序规模较大时，自动机状态超出了处理能力范围。模型检测一个重要的问题是设计好的剪枝算法和数据结构使得可以搜索尽可能大的状态空间，以验证更大规模的系统。MOPS由于采取了路径不敏感分析、属性分解等技术，可以验证Linux内核规模（百万行代码级别）的程序[[Schwarz, Chen et al. 2005](#_ENREF_43)]，但同时也带来了误报的问题。BLAST采用了懒惰抽象[[Henzinger, Jhala et al. 2002](#_ENREF_30)]和反例引导逐步求精[[Clarke, Grumberg et al. 2000](#_ENREF_15)]的技术，从比较粗糙的模型出发，当发现反例路径时精华相关部分程序模型，因而扩展了适应规模。

### 定理证明

定理证明 (theorem proving)是一种形式验证方法，它是根据已构造的规格说明生成反映该规格说明应具有的性质，将其表示成定理形式，并加以证明，从而达到对系统规格说明验证的目的。形式化的方式主要思路是对程序源代码、程序规范说明进行建模、分析，以判断程序是否符合程序规范的要求，并对不符合要求的程序给出反例路径，供开发人员分析修正。定理证明的逻辑性强，证明理论比较复杂，有时候需要用户进行干预、指导，阅读证明的过程对人员要求较高。

使用定理证明验证程序的做法，通常是使用抽象解释的方法对程序进行转换，转换为一系列的变量间的约束关系，然后用定理证明器、可满足性判定器[[Riazanov and Voronkov 2002](#_ENREF_42); [Detlefs, Nelson et al. 2005](#_ENREF_24); [Barrett and Tinelli 2007](#_ENREF_4); [Conchon and Contejean 2008](#_ENREF_19); [Moura and Bjørner 2008](#_ENREF_37)]来验证这些约束关系的成立与否。定理证明是程序验证中常用到的技术，已有的证明器其求解效率和自动化程度已经可以被人们接受，因而它得到了广泛的应用。

谓词抽象

## 难点

形式化证明的一个关键难点在于程序中循环结构的存在，而且循环中常常又有嵌套循环和分支结构。这导致了模型检测中的状态空间爆炸问题，也导致了抽象解释中对变量值范围的过度放大引起的不准确问题。不能准确把握循环结构对程序状态的影响，为证明过程提供充分的辅助信息，使得证明不能顺利完成[[Moy and Marché 2010](#_ENREF_38)]；另一个难点在于程序中数组结构的使用。数组的大小和存储的元素通常是不确定的，对数组的处理通常又出现在循环结构之中，这为准确分析数组中各个元素值带来了困难。递归程序。

目前对循环的处理方法主要有。

单一顺序执行

循环展开

路径程序

单一

综上，循环不变式是循环的后置条件的一种弱化形式[[Furia and Meyer 2010](#_ENREF_26)]。某个程序规范未得到证明，不一定就是因为程序和规范不一定，而是因为证明需要的信息不充分，没有给出足够的程序中包含的循环不变式。这表明，循环不变式的重要性[[Moy and Marché 2010](#_ENREF_38)]

# 第三章 循环不变式生成技术研究

上一章阐述了形式化方法的基础理论，并指出程序证明过程的一个关键问题是寻找循环结构中存在的不变式，这正是本文研究的重点。本章将从几个具体程序出发，根据其具体特点，分析了现有的生成循环不变式的方法，指出了它们的优缺点。在这些现有方法的基础上，有针对性地提出了改进目标。

基于迭代的不动点计算：数据流分析，模型检测，抽象解释等

基于约束的不变式生成：目标指向性明确，效率高。不需要放大，精确。[[刘自恒报告20110301, Gulwani et al. 2008](#_ENREF_47)]

针对线性算术式的不变式模板求系数法；[[刘自恒报告20110301, Gulwani et al. 2008](#_ENREF_47); [Gupta and Rybalchenko 2009](#_ENREF_28)]由用户指定不变式模版（程序有默认的可以使用），是基于数值约束求解[[Colon, Sankaranarayanan et al. 2003](#_ENREF_18)]的。对于此，用户指定的模板形式固定，适应性不强，可以通过要证明的规范或断言，启发式地得到可能的不变式模板形式，然后逐一进行约束求解。约束求解库可以使用APRON，定义的抽象域包括Box,Oct，NewPolka，PPL等，不同的抽象域求解的效率和精度不同。[[Gupta and Rybalchenko 2009](#_ENREF_28)]将自己集成到Blast中，使Blast的分析效果得到提升。

Interproc使用APRON和fixpoint库实现了一个简单语言程序的不变式生成，fixpoint是按照[[Gopan and Reps 2007](#_ENREF_27)]中的思想实现的，不同的转换系统实例可以得到不同的分析结果，[[Gopan and Reps 2007](#_ENREF_27)]中给出了两个转换系统实例。可以将Interproc看作一个抽象解释器，invGen即在前端使用了Interproc作为解释器，并跟踪它生成的不变式。

多阶段循环的展开法(循环展开，控制流精化)或条件赋值法；当循环体中有if语句时，程序的路径有多条，由于循环控制变量的变化，可以把循环看成一个多阶段的执行过程。可以将循环体做语句展开，做执行路径的具体化，或者将if控制下的语句转换为条件赋值语句[[刘自恒报告20110426, Kovács et al. 2009](#_ENREF_48)]，或者通过分割谓词将一个循环展开为多个基本形式的循环[[Sharma, Dillig et al. 2011](#_ENREF_44)]。[[Beyer, Henzinger et al. 2007](#_ENREF_6)]将这样的循环连同其他部分看做多个路径程序分别分析。将if条件直接转换到不变式中，同时做相应的变量转换，类似[[Furia and Meyer 2010](#_ENREF_26)]，之前黄老师说不加改变的是无意义的，因为在推理的时候，推理证明器可以根据if条件进行推理。

将数组根据状态划分为小数组的分割法；有数组参与时，可以将所有数组元素用互不同的变量代替，也可以用一个变量代替所有元素，或根据循环的执行进程，将数组分为不同状态的子数组[[刘自恒报告20110426, Kovács et al. 2009](#_ENREF_48); [Cousot, Cousot et al. 2011](#_ENREF_22)]，根据状态得到数组变量的不变式性质。有字符串操作的语句也可以看成对数组进行操作，一般通过系统函数如memcpy等完成，这时候可以对系统函数加上规范说明得到关于字符数组的不变式性质。

[[Flanagan and Qadeer 2002](#_ENREF_25)]使用了预定义原子谓词来辅助程序证明过程，使用谓词抽象技术猜测后面可能使用到的谓词。

# 第四章 循环不变式生成技术改进

第三章总结了现有的生成循环不变式的方法，并指出了它们的适用范围与不足。本章提出了一种改进的不变式生成技术，主要基于条件赋值转换和自适应模板生成方法。

对于数组操作的情形

数值计算程序

# 第五章 loopInv插件的设计与实现

本章介绍第四章设计的循环不变式生成技术的具体实现。该实现是基于Frama-C平台和APRON库的，作为Frama-C的一个插件而集成到Frama-C平台中，命名为loopInv，并提供编程接口供其他开发使用。本章首先简要介绍相关的Frama-C平台和APRON库的相关内容，然后给出loopInv插件的结构组成和处理流程。

平台介绍

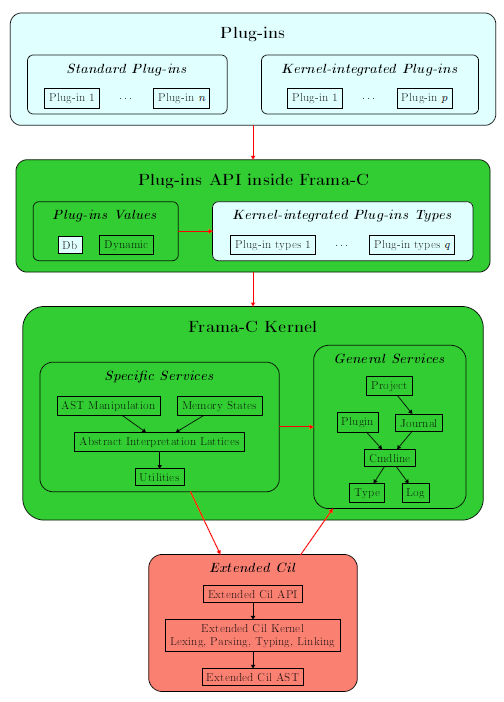
ACSL

ACSL的全称是ANSI C Specification Language，是一种在Frama-C框架下实现的行为接口类规范描述语言（Behavioral Interface Specification Language，BISL），用来描述C语言代码的行为属性。不同于JML（Java Modeling Language）既用于动态检测断言又用于静态验证，ACSL用于静态验证和归纳验证。

更多内容见<http://frama-c.com/acsl.html>。

Frama-C

Frama-C是一个用来执行C程序静态分析的工具和开发平台，以CIL[[Necula, McPeak et al. 2009](#_ENREF_39)]为前端预处理工具，提供了可扩展的插件机制。它收集了多种静态分析技术，如依赖分析、值分析、抽象解释等，将它们作为插件整合在一个统一的框架中，并对外提供开发接口，允许用户调用既有插件的结果，达到快速开发自己的功能插件的目标。**图 5.2**是Frama-C的结构设计图。



**图 5.2 Frama-C结构图**

APRON

提到，现在有多种类型的抽象域模型，但一直以来缺乏一个统一、一致的实现，其他程序调用时不能方便地在不同抽象域之间切换。Bertrand Jeannet等人完成了这项工作，实现了一个名为APRON的函数库。

获取循环中可能的变量值更新变化的步长集合

获取循环中更新变量值的常数集合

获取循环之后程序中存在的判断表达式集合Adj

提取Adj中的变量集合。通过插件PDG进行依赖关系计算，得到循环中使用每个变量的结点集合。提取其中的常数，作为步长候选值。

根据循环中用的变量，生成模板，使用wp验证是否满足，不确定时加入转换系统，由APRON再次检查。

成立的不变式用ACSL注释表示，加入程序中，输出得到分析后的含有循环不变式的程序。使用WHY等工具来验证程序中的属性是否可以得到满足。

# 第六章 实验过程与结果

**表 6.1 比较**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 第七章 结论与展望

由于理论尚不够成熟，形式化方法还难以证明大型软件系统的正确性[[Airchinnigh 1995](#_ENREF_1)]。希望我们的工作有助于形式化证明的发展。

# 参考文献

# Airchinnigh, M. a. (1995). Formal Methods & Testing. Tutorials of the Sixth International Software Quality Week, 625 Third Street, San Francisco, CA 94107-1997.

# Ball, T., B. Cook, et al. (2004). SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Proceedings of Fourth International Conference on Integrated Formal Methods, Springer.

# Ball, T. and S. K. Rajamani (2002). The SLAM project Debugging system software via static analysis. Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon, ACM New York, NY, USA ©2002.pp: 1 - 3.

# Barrett, C. and C. Tinelli (2007). CVC3. CAV '07, Springer, July 2007.pp: 298-302.

# Beyer, D., T. A. Henzinger, et al. (2007). "The Software Model Checker Blast: Applications to Software Engineering." International Journal on Software Tools for Technology Transfer 9(5-6): 505-525.

# Beyer, D., T. A. Henzinger, et al. (2007). Path invariants. PLDI '07, ACM New York, NY, USA ©2007.pp: 300 - 309.

# Bradley, A. R. and Z. Manna (2008). "Property-Directed Incremental Invariant Generation." Formal Aspects of Computing 20(4-5): 379–405.

# Chaki, S., E. Clarke, et al. (2003). Modular Verification of Software Components in C Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon.

# Chen, H., D. Dean, et al. (2004). Model Checking One Million Lines of C Code. Network and Distributed System Security Symposium.

# Chen, H. and D. Wagner (2002). MOPS an infrastructure for examining security properties of software. Proceedings of the 9th ACM conference on Computer and communications security, Washington, DC, USA.pp: 235-244.

# Chen, L., A. Mine, et al. (2008). A sound ﬂoating-point polyhedra abstract domain. APLAS 2008.pp: 3-18.

# Cimatti, A., E. Clarke, et al. (2002). NuSMV 2: An OpenSource Tool for Symbolic Model Checking. CAV'02, LNCS(2002).pp: 241-268.

# Cimatti, A., E. Clarke, et al. (1999). NuSMV: A New Symbolic Model Verifier. CAV'99, LNCS.pp: 495–499.

# Clariso, R. and J. Cortadella (2004). The octahedron abstract domain. SAS'04.pp: 312-327.

# Clarke, E., O. Grumberg, et al. (2000). Counterexample-Guided Abstraction Reﬁnement. CAV'00.pp: 154–169.

# Clarke, E. M. and E. A. Emerson (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs, Springer-Verlag London, UK(1982).pp: 52-71.

# Clarke, E. M., O. Grumberg, et al., Eds. (2000). Model Checking, The MIT Press.

# Colon, M., S. Sankaranarayanan, et al. (2003). Linear invariant generation using non-linear constraint solving. CAV'03.pp: 420-433.

# Conchon, S. and E. Contejean (2008). The Alt-Ergo automatic theorem prover.

# Cousot, P. and R. Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Principles of programming languages. New York, ACM Press: 238–252.

# Cousot, P. and R. Cousot (1979). Systematic design of program analysis frameworks. POPL'79, ACM, New York, NY, USA(1979).pp: 269 - 282.

# Cousot, P., R. Cousot, et al. (2011). A parametric segmentation functor for fully automatic and scalable array content analysis. POPL'11, ACM.pp: 105-118.

# Cousot, P. and N. Halbwachs (1978). Automatic discovery of linear restraints among variables of a program. POPL'78, ACM Press, New York (1978).pp: 84-96.

# Detlefs, D., G. Nelson, et al. (2005). "Simplify: a theorem prover for program checking." Journal of the ACM (JACM) 52(3): 365-473.

# Flanagan, C. and S. Qadeer (2002). Predicate Abstraction for Software Verification. POPL '02.pp: 191–202.

# Furia, C. A. and B. Meyer (2010). Inferring Loop Invariants Using Postconditions, Springer-Verlag Berlin, Heidelberg ©2010

# Gopan, D. and h. Reps (2007). Guided Static Analysis. 14th International Static Analysis Symposium, Kongens Lyngby, Denmark.pp: 349--365.

# Gupta, A. and A. Rybalchenko (2009). InvGen: An Efficient Invariant Generator. CAV '09.pp: 634–640.

# Havelund, K. and T. Pressburger (2000). "Model checking JAVA programs using JAVA PathFinder." INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER (STTT) 2(4): 366-381.

# Henzinger, T. A., R. Jhala, et al. (2002). Lazy abstraction. Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon.pp: 58-70.

# Henzinger, T. A., R. Jhala, et al. (2003). Software verification with BLAST. Proceedings of the 10th international conference on Model checking software, Springer-Verlag Berlin, Heidelberg ©2003.pp: 235-239.

# Holzmann, G. J. (1997). "The Model Checker SPIN." IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 23,n. 5: 279-295.

# Jeannet, B. and A. Miné (2009). APRON: A Library of Numerical Abstract Domains for Static Analysis. CAV'09.pp: 661-667.

# Manna, Z. and A. Pnueli (1995). Temporal Verification of Reactive Systems: Safety, Springer.

# Mauborgne, L. (2004). Astrée: verification of absence of run-time error. Building the Information Society, Kluwer Academic.pp: 385—392.

# Miné, A. (2006). "The Octagon Abstract Domain." Higher-Order and Symbolic Computation 19(1): 31 - 100.

# Moura, L. d. and N. Bjørner (2008). Z3: An Efficient SMT Solver. TACAS'08, Springer, Heidelberg (2008).pp: 337–340.

# Moy, Y. and C. Marché (2010). "Modular inference of subprogram contracts for safety checking." Journal Of Symbolic Computation 45(11): 1184-1211.

# Necula, G., S. McPeak, et al. (2009). CIL: Infrastructure for C Program Analysis and Transformation.

# Necula, G. C., S. McPeak, et al. (2002). CIL: Intermediat e Language and Tools for Analysis and Transformation of C programs. Conference on Compiler Construction.pp: 213–228.

# Queille, J.-P. and J. Sifakis (1982). Specification and verification of concurrent systems in CESAR. Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag London, UK(1982).pp: 337 - 351.

# Riazanov, A. and A. Voronkov (2002). "The design and implementation of VAMPIRE." AI Communications 15(2-3): 91-110.

# Schwarz, B., H. Chen, et al. (2005). Model checking an entire linux distribution for security violations. ACSAC '05.pp: 13-22.

# Sharma, R., I. Dillig, et al. (2011). Simplifying Loop Invariant Generation Using Splitter Predicates. CAV'11.pp: 703-719.

# Simon, A., A. King, et al. (2003). Two variables per linear inequality as an abstract domain. LOPSTR 2002, LNCS.pp: 71–89.

# V.Aho, A., M. S.Lam, et al. (2007). Compilers Principles Techniques and Tools. America, Addison Wesley.

# 刘自恒报告20110301, S. Gulwani, et al. (2008). Program analysis as constraint solving. PLDI '08, Tucson, Arizona, USA, ACM New York, NY, USA ©2008.pp: 281 - 292.

# 刘自恒报告20110426, L. Kovács, et al. (2009). Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. FASE 2009, Springer-Verlag Berlin Heidelberg 2009.pp: 470–485.

# 攻读硕士学位期间参与的科研项目

# 攻读硕士学位期间完成的论文

# 致谢

在此感谢。。。